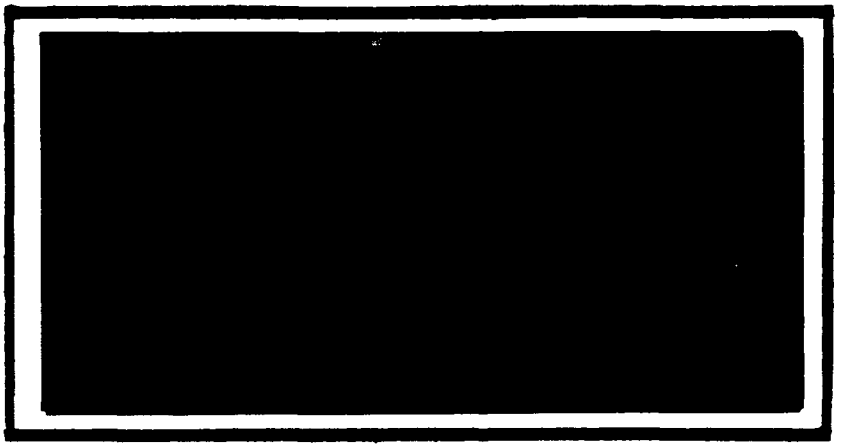
(1)



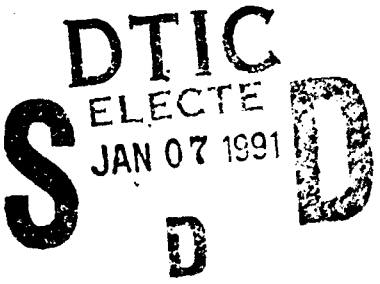
DEPARTMENT OF THE AIR FORCE

AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GCS/ENG/90D-16



A HYPERMEDIA IMPLEMENTATION FOR REUSABLE SOFTWARE COMPONENT REPRESENTATION

THESIS

Gary G. Worrall
Captain, USAF

AFIT/GCS/ENG/90D-16

AFIT/GCS/ENG/90D-16

A HYPERMEDIA IMPLEMENTATION FOR REUSABLE SOFTWARE COMPONENT REPRESENTATION

THESIS

Presented to the Faculty of the School of Engineering
of the Air Force Institute of Technology
Air University
In Partial Fulfillment of the
Requirements for the Degree of
Master of Science (Computer Science)

Gary G. Worrall, A.A.S., B.S.C.S.
Captain, USAF

December, 1990

Preface

The purpose of this study was to investigate the use of hypertext to implement the representation of reusable software components. Several representation methods have been proposed in the literature and several have been implemented in real or, more typically, experimental systems. Hypermedia, with its capability to represent knowledge in a non-linear manner appears to offer benefits for a wide range of potential software reusers.

Results from this and similar studies are important for libraries of reusable software components to effectively meet the needs of potential users. Without a clear understanding of the benefits and drawbacks of possible representation methods, designers of software reuse libraries cannot be expected to create truly useful systems.

I am deeply indebted to my faculty advisor, Maj David Umphress, for his great patience, gentle prodding, and willing assistance as I worked on this thesis. Thanks also to my thesis committee members, Prof. Dan Reynolds and Maj Marty Stytz. Finally, I wish to express my thanks and love to my wife, Andrea, and my sons, Colin and Graham, for their love, support, and understanding during the trying times when I was so wrapped up in this work that they were virtually ignored.

Gary G. Worrall

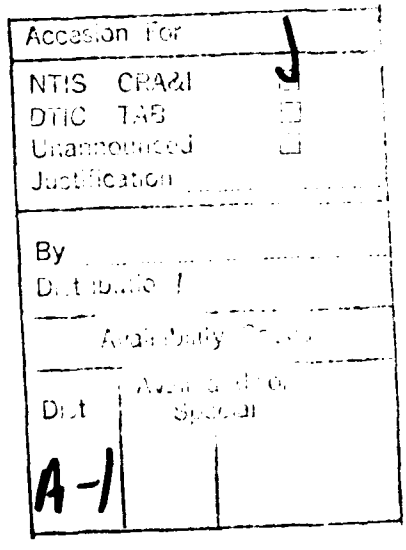


Table of Contents

List of Figures

List of Tables

Abstract

This study investigated representation methods for software reuse. Hypermedia was chosen as the implementation method to represent a collection of reusable software components in a Reusable Software Component Representation System (RSCRS). The hypermedia implementation organizes knowledge about the component collection, and individual components, into a web of small information chunks called frames.

The set of software components was represented within RSCRS using a hybrid classification scheme composed of an enumerated part and a faceted part. The enumerated part of the classification system enables the user to progress along a path in a taxonomic tree, successively narrowing the scope of eligible components. Each leaf node in this tree denotes a class of reusable software components, members of which are distinguished by their time and space characteristics. These characteristics are grouped into eleven facets, each of which is comprised of two, three, or four elements known as forms.

Explicit links between frames establish a domain dependent means of traversing the information net. Some of these links allow the user to progress directly through the levels of the classification structure. Other links lead from the classification structure frames to frames containing explanatory text for the terms used in the classification. Additional links serve as cross-references between related topics.

A simple component locator is provided which utilizes information from the user's frame selections to identify, and provide locating information for, a set of components meeting the user's selection. To round out the RSCRS system, a component source code browsing capability is provided for the selected components.

A HYPERMEDIA IMPLEMENTATION FOR REUSABLE SOFTWARE COMPONENT REPRESENTATION

I. Introduction

> Human progress has resulted from one generation building upon the successes and failures of previous generations. In most fields of engineering, practitioners follow standard models, and deviate from a standard practice only when the project at hand imposes a special requirement. When software designers address a new project or a major upgrade, however, the dominant practice is to begin anew, or at best, to focus only on the prior experience of the immediate team of players. The result is a proliferation of "unique" software that is functionally identical. This duplication includes not only code, but also design, test programs, data and instrumentation, and maintenance. (Wald, 1987:353)

> ... we should stand on each other's shoulders rather than on each other's feet ... (Wegner, 1983:33)

> We would not retain a software engineer who designed, coded, tested, and documented a new sine routine every time a new system required that mathematical function; we would expect instead that a library function would be reused. Yet we tolerate and even expect that more demanding functions will be redeveloped in each new system. (Wald, 1987:353)

1.1 Problem Statement

The method used to represent reusable software components in software reuse libraries is a key factor affecting the ease of use and effectiveness of such libraries.

An understanding of the diverse approaches available for implementing software component representations will aid in the wise selection of component representation methods and their implementations for software reuse projects.

1.2 Background

The demand for computer software is increasing faster than the software engineering community's productivity. The cost to develop and support computer software is a major factor affecting system acquisitions within the United States Department of Defense (DOD).

Much newly developed software duplicates existing software. Horowitz and Munson relate a study which "observed that 40–60 percent of actual program code was repeated in more than one application" (Horowitz and Munson, 1984:478). "...60 percent of all business application designs and code are redundant and can be standardized and reused" (Lanergan and Grasso, 1984:498). Jones mentions an unpublished study which noted that about 75 percent of functions in commercial banking and insurance applications were common ones that occurred in more than one program (Jones, 1984:488). He reaches the tentative conclusion "that of all the code written in 1983, probably less than 15 percent is unique, novel, and specific to particular applications" (Jones, 1984:488).

This duplication of effort is costly: productivity and reliability are not what they could be. Most work in the area of software reuse is based on the assumption, often implicit, that software reuse may provide the key opportunity to escape the escalating lifecycle cost spiral by increasing the productivity of software developers, reducing the costs of system development, improving the reliability of software systems, and reducing the requirements for software support (Booch, 1987:6; Margono and Berard, 1987:63; Sommerville, 1989:353). Reuse can increase the software developer's capabilities by allowing him or her to generate equivalent or better products with less work (Anderson, 1988:3). Reuse should reduce the number of components

to be specified, designed, implemented, and tested which should lead to a shorter schedule and thus to reduced cost (Tracz, 1987:358). But, "the approach based on reusable components has not been tested in practice although it appears to offer advantages for both costs and system reliability" (Sommerville, 1989:7).

Despite the fact that software reuse is currently a very active research area, it is not at all new. For example, code reuse was the main motivation for early subroutine libraries: it is only recently that subroutine libraries received their current emphasis as program structuring concepts (Sommerville, 1989:352).

"Reusability is a general engineering principle whose importance derives from the desire to avoid duplication and to capture commonality in undertaking classes of inherently similar tasks" (Wegner, 1984:9). Modern software reuse can be viewed "as analogical development, where a previous development from similar requirements is transformed into a new development satisfying new constraints" (Perry, 1988:1). In other words, software reuse is "the use of 'sequences' of software development 'solutions' from previous projects to 'solve' a current development" (Perry, 1988:1). Figure 1 illustrates this view of software reuse.

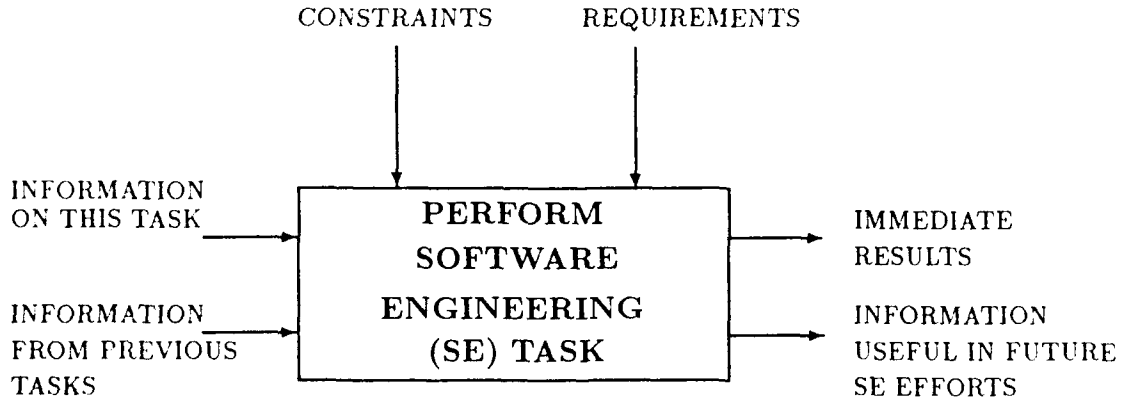


Figure 1. Central Idea of Reusable Software Engineering (Freeman, 1987:830)

Thus, software reuse is the use of previously acquired concepts and objects in a new software development situation (Prieto-Diaz and Freeman, 1987:7). On one level, the reuse of previously acquired concepts includes the reuse of ideas and knowledge such as general software design principles for determining the size and performance characteristics of a component. This is analogous to the reuse of concepts in other engineering disciplines. For example, aeronautical engineers apply standard design equations to determine the dimension and materials of a wing. On a separate level, the reuse of objects encompasses the selection of an existing software component to meet specified design criteria. This is analogous to the reuse of objects in other engineering disciplines. For example, electrical and mechanical engineers consult parts catalogs to identify the available parts which best fit the design constraints.

Viewed in this manner, software reuse is much more than simple reuse of actual code; it includes reusable design, various forms of specification systems, application generators, and prototyping systems (Horowitz and Munson, 1984:477). "Any lifecycle product falls within the scope of the reuse problem" but, "little is known about the reuse of software other than code. Reuse, then, includes such lifecycle objects as: concept documents, estimates, requirements, designs, code, test plans, maintenance plans, and user documentation" (Frakes and Gandel, 1989:302).

1.3 A Framework

The foreword to the special software reuse issue of *IEEE Transactions on Software Engineering* provides a framework for organizing the various approaches to software reuse (Biggerstaff and Perlis, 1984). This framework appears in the literature again two and a half years later (Biggerstaff and Richter, 1987). The framework, summarized in Table 1, divides the various approaches into two groups based upon the nature of the components being reused.

Features	Reusability approaches				
Component reused	Building blocks		Patterns		
Component nature	Atomic and immutable		Diffuse and malleable		
	Passive		Active		
Reuse principle	Composition		*Generation*		
Emphasis	Application component	Organization and composition principles	Language-based	Application generators	Transformation systems
Typical systems	Subroutine libraries	Object-oriented	Very high-level languages	Terminal-display formatters	Language transformers
		Pipe architectures	Problem-oriented languages	File management	

Table 1. A Framework For Reusability Technologies (Biggerstaff and Richter, 1987:42)

The first group, composition technologies, includes atomic, and relatively immutable, passive building blocks which are used to derive new programs by applying a few well-defined composition principles. Examples of these building blocks are code skeletons, subroutines, functions, programs, and objects. Proponents of this approach assume that systems are primarily composed of components which already exist so the system development process becomes one of assembly rather than creation (Sommerville, 1989:7). Work in this group emphasizes two classes: application component libraries themselves and principles of component organization and composition.

The other group, generation technologies, deals with reusable patterns as active elements which generate the target programs. This group emphasizes three classes: language-based systems, application generators, and transformation systems. Language-based systems achieve reuse via very high-level languages (VHLL) and problem-oriented languages. Application generators include systems such as terminal-display formatters and file managers. Transformation systems reuse abstract programs by transforming the VHLL description of an abstract component to the implementation language of its concrete counterpart (Cheatham, 1984).

The boundaries between these three classes are not very clear. Several systems which obviously belong in this group exhibit characteristics of all three classes. Example generation technology systems include the Constructor Function within the Parts Engineering System (PES) of the Common Ada Missile Packages (CAMP) (Anderson, 1988:3), the Draco system (Neighbors, 1984), and the TAMPR LISP to FORTRAN program transformation system (Boyle and Muralidharan, 1984).

This thesis concentrates on the first group — code building blocks. Reusable code components can be divided into three classes:

- plug compatible: these components can be used as-is. Assuming they are guaranteed to perform as specified, they offer a quick way to begin assembling a system. However, these components are specifically constructed to apply to a particular object type.
- parameterized (generic): these components are derived from templates via instantiation with specific parameter values. They offer more flexibility than plug compatibles because they can be instantiated for different object types needing similar functionality.
- adaptable: These components are not reusable as-is, but can either be adapted or permit usable code to be abstracted from them. Thus they may save design and coding time over building from scratch.

(Knapper, 1988:234)

Although reuse of logical components in any form seems to have obvious benefit, some have argued that code reuse alone can never be cost effective (Sommerville, 1989:352). One of the clear successes in code reuse is the reuse of numerical computation routines. However, numerical computation is unique in several ways:

- The domain is very narrow, containing only a small number of data types. This narrowness makes code reuse more manageable; because there are only a few data types, these components have a higher probability of reuse.

- The domain is well understood, its mathematical framework having developed over hundreds of years. Thus the investment required to create a library of reusable code components is reduced. Since many people understand the domain, they may readily understand a component's function with only a brief functional description.
- The underlying technology is quite static, growing and evolving very slowly. Importantly, it evolves so existing parts of the technology remain unchanged yielding upward compatibility of the technology. Thus, the component library remains relatively stable, allowing the using organization to amortize its cost over a longer period of time.

(Biggerstaff and Richter, 1987:44)

Reuse of software components appears to offer the potential for reduced development costs, improved reliability, and ultimately lower lifecycle costs. But, a number of impediments to effective software reuse exist: acquisition, technical, and managerial (Cardow, 1989:564). One particularly crucial technical issue is the design of *software reuse libraries*.

Two reasons why a programmer doesn't use someone else's code or design are:

- It is easier to write it oneself, then (sic) to try to locate it, figure out what it does, and find out if it works. ...
- There are no tools to help find components or compose a system from the reusable pieces.

(Tracz, 1987:359)

Many researchers agree with this observation as evidenced by the following quotations.

Frequently, software is not reused because the *value* of reusing software is low and the *feasibility* of reuse is minimal. ...if the retrieval and

> specification of software components are not automated, the amount of time required to locate reusable software increases greatly. If potentially reusable software components cannot be located, retrieved, and reviewed effectively, reuse is neither feasible nor valuable.
>
> ⋮
>
> In short, there is strong support for the belief that to effectively promote software reuse we must develop tools to aid in the process of locating software components that are candidates for reuse. Such a tool must provide the user with an effective means of indexing, searching, retrieving, and reviewing software components. (Frakes and Nejmeh, 1987:381)

Thus,

> - a software component can be effectively reused only if it includes a large amount of information describing different aspects of the component, such as its behaviour, the problem it faces, the constraints limiting its use and so on.
> - a programmer will reuse a component developed by others only if the time needed to properly instantiate it is less than the time required to develop a new component from scratch.
>
> Unfortunately the first principle is in conflict with the second, because the information associated to the component often requires a lot of time to be read and understood by the programmer. To solve this conflict, the programmer must be supported through "intelligent" techniques in understanding the information associated to a component. These techniques should be able to present the additional information in a smart way, not only as simple sequential text. (Ghisio and others, 1987:386)

A software reuse library functions primarily to enable the software developer to locate appropriate software components for use in the developer's particular application. As can be seen from the above quotations, ease of use and effectiveness are essential to successful software component libraries. A key factor affecting ease and effectiveness of reuse is the language chosen to describe the set of software components. This descriptive language, which may be textual or graphical, is referred to as a representation.

The reason that a representation is created is to allow operations on the representation that would be more difficult or impossible on the represented object itself. It is much easier, for example, to sort a set of bibliographic records according by author than to sort the same number of books by author.

⋮

A representation system has three levels as shown in [Table 2]. The

Level	Example
Presentation	Graphical Tree
Representation	Class Hierarchy
Implementation	Paper Manual

Table 2. Representation System Levels

presentation is what the user of the system actually sees when using the system. The representation is the logical model, and the lowest level is the way the system is actually implemented where the implementation might be a printed manual or an automated data base system of some sort. A lower level in this scheme constrains the levels above it. (Frakes and Gandel, 1989:303)

1.4 Scope

For this thesis effort, selected representation methods were applied, in a prototype Reusable Software Component Representation System (RSCRS), to an existing library of reusable software components.

1.5 Overview of This Thesis

This chapter has presented a general discussion of software reuse. Chapter II continues the discussion with a survey of the literature specifically addressing software component representation, presentation, and implementation methods. Chapter III then discusses the specific methods used in developing the RSCRS. Chapter IV

concludes this thesis with a summary of the work performed, the conclusions to be drawn from this work, and recommendations for further research.

II. Review of Previous Work

This chapter provides a review of the literature pertaining specifically to representation methods for reusable software components. The first section presents the concept of domain analysis. The problem domain addressed by a collection of software components has a direct impact on the appropriateness of the method or methods used to represent the components. The second section presents the notations and methodologies which have been proposed for software component representation. These notations and methodologies are drawn from the fields of library and information science, computer science, and discrete mathematics. The third section presents a brief overview of some of the existing collections of reusable software components.

2.1 Domain Analysis

The application domain is important to any discussion of software reuse. For example, consider the database retrieval/process/report stereotype. This application domain is ideally suited to application generators via control language, a concept which has been very effective but has not spread widely to other domains. There seem to be fewer obvious stereotypical situations in other domains, thus less scope for application generators.

"Domain analysis is the process of deriving a domain model of a given software system" (Frakes and Gandel, 1989:303). Although the suitability of application generators may be limited, the domain analysis step is important for software reuse. For example, domain analysis is the most important aspect of Draco, one of the generation technology systems mentioned in Chapter I.

In Draco, a specific problem domain is analyzed resulting in a new domain language. This approach allows reuse of both analysis (the *what*) and design (the

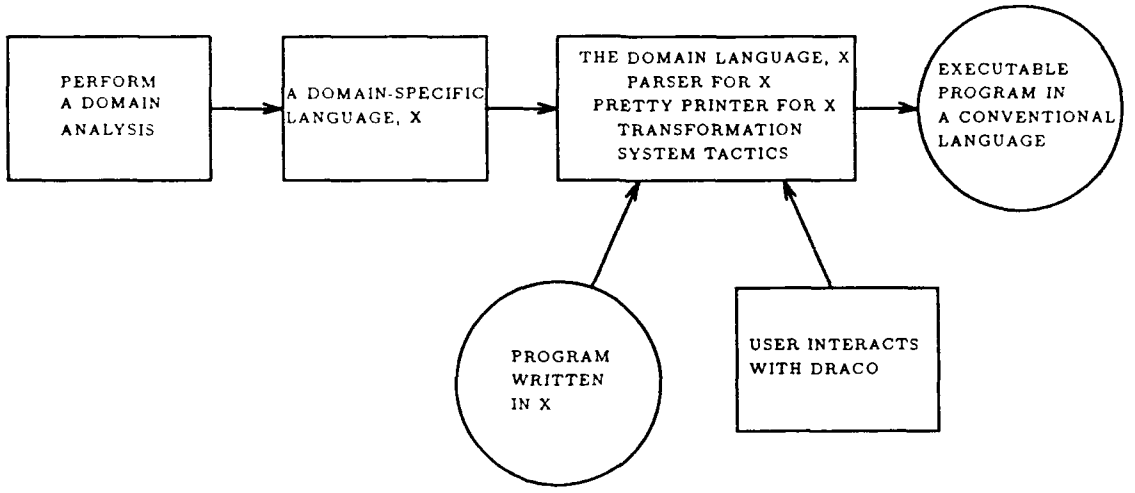


Figure 2. Schematic for Draco (Horowitz and Munson, 1984:482)

how) information. This reuse is accomplished through source-to-source transformations, between domain languages, on objects and operations in the domain. Each component, then, consists of one or more refinements, where each *refinement* is a statement of the problem semantics in terms of one or more of the domain languages known to Draco (Neighbors, 1984). Figure 2 presents the steps involved in the Draco process.

In any reuse effort, domain analysis provides the knowledge necessary to determine appropriate component classification and representation methods. For example, the Common Ada Missile Packages (CAMP) project examined the tactical air-launched missile domain. The results of the domain analysis were used to develop a library of reusable components and a limited capability to derive new components from existing components based on specifications (Anderson, 1988).

2.2 Notations and Methodologies

A comparison between software development and hardware development shows that hardware developments use many existing components with the minority of

system components being specially built. If software component reuse becomes a widespread reality, the main emphasis in programming will shift from program development to component interconnection. This shift implies a need to develop new notations and methodologies. These new notations would concentrate on describing system structure in terms of its basic components and ignore the details of how the components operate (Sommerville, 1989:310).

> Issues to be considered when selecting a representation for reusable components fall into three major categories: issues about the objects to be represented, issues about the system of objects to be represented, and issues about the representation itself (Frakes and Gandel, 1989:303).

> When a component is reused, implications on and from the environment must be known and considered during the instantiation process. Other information like instancing constraints, component user manual and component description must be supplied and used for a correct reuse of the component. (Ghisio and others, 1987:386)

"...our methods for determining the definitions of components are weak, it is difficult to describe a component to a user, and there are no tools to catalog, refine, and compose components in an efficient manner" (Horowitz and Munson, 1984:481).

Careful selection of components is key to successful software reuse. Software repositories or libraries provide a means of cataloging, storing, and accessing reusable components. No standardized method exists for classifying, cataloguing, and retrieving software components. To obtain maximum benefit from a software library it must be readily accessible, easy to use, and oriented toward user requirements. The actual storage and retrieval mechanism should be invisible to the user. For reuse to become truly practical on a large scale, the cost to retrieve a component must be far less than the cost to develop the equivalent component.

> We must learn how to organize, index, describe, and reference software components effectively. We believe that a system of "software component folders" could be organized and indexed by conventional techniques for indexing papers in the computer science literature, and that by having each component in a software library in a form susceptible to parametric variation and refinement, an effective solution to this problem could be found. (Standish, 1984:496)

> The techniques that can be provided to support information understanding could be roughly divided into two classes:
>
> 1. techniques which handle the semantics of the additional information;
> 2. techniques which handle only the structure of the additional information.
>
> Both types of techniques are necessary to give a suitable support in component understanding. (Ghisio and others, 1987:386)

2.2.1 Indexing. Indexes communicate knowledge to users about information items through specialized indexing languages. An indexing language summarizes or describes the subject or content of information items. An indexing language consists of three parts. The first part is the terms (or elements) that make up the language. The second part is the syntax (or rules for combining terms). The third part is the semantics (or logical relationship between terms). (Frakes and Gandel, 1989:304-305)

Indexing languages can be classified along a continuum where controlled vocabularies and uncontrolled vocabularies are the endpoints of the continuum. Controlled vocabularies are those vocabularies that limit the terms that can be selected, limit the ways terms can be synthesized, or limit the semantic relationships between terms. Uncontrolled vocabularies are those vocabularies that place little, if any, restraint on term selection or synthesis, and do not limit the semantic relationships between terms. (Frakes and Gandel, 1989:304-305)

2.2.1.1 Controlled Vocabularies.

The purpose of a controlled vocabulary is:

- To promote consistent representation of subject matter by indexes and searches.
- To simplify comprehensive searching on a topic by linking together items with related meanings.

Generally, terms for controlled vocabularies are derived usi: g a combination of two methods.

- Literary Warrant — index terms are derived from the examination of the subject area.
- User Warrant — index terms are included if it is of interest to the user population.

There are two major forms of controlled vocabularies — classed systems and keyword systems. (Frakes and Gandel, 1989:305)

Classed Systems. The traditional library science method of creating a representation is known as indexing or classification. Classification is the process of distinguishing components which possess a certain property or characteristic from those that lack it and grouping components which have the property or characteristic in common into a class or category. Classification "is concerned with not only the relationship between things but also the relationship between classes of things" (Frakes and Gandel, 1989:305). Classification schemes range along a continuum with a strictly enumerative approach at one extreme and a completely faceted approach at the other extreme.

There are two general types of classification schemes; enumerative and faceted. Enumerative schemes take a subject area and divide it into successively narrower classes listing all the elemental, superimposed, and compound classes arranged in order of their hierarchical relationships. This "listing" of all possible subjects is the major disadvantage of the enumerative scheme [Buchanan 1979]. Any subject (class or subclass)

which does not appear in the schedule cannot be classified (located in the collection).

The second type of classification scheme, faceted, is more of a building block approach emphasizing subject analysis and synthesis [Chan 1981]. An analysis process is used to construct the classification schedule. Subjects to be classified are analyzed and divided into their elemental terms (e.g., things defined by only one characteristic). Only the elemental terms and their relationships are listed in the faceted schedule. Recurring divisions are not repeated in each major class as they are in the enumerated scheme. The elemental terms are listed separately for application to all subjects as needed. Synthesis is then used to express a superimposed, complex, or compound class by assembling its elemental parts from the facets according to the citation order. Since the analysis and synthesis process plays such a major role, faceted systems are also called analytico-synthetic classification in the literature [Chan 1981; Buchanan 1979; Vickery 1960]. (Ruble, 1987:50)

The major advantage of using enumerated classification systems is their structure. The well defined hierarchical structure makes it easy for the user to interpret the relationship among terms. Users can easily modify their searches to be more specific or more general by moving down or up the hierarchical tree. The disadvantage of this classification system is its very strength. Its top down approach requires an exhaustive analysis of the area for which it is developed. It is also a rigid structure that can support only one view of the relationship between elements. This rigidity makes it very hard to change an enumerated classification system without restructuring the whole hierarchy — except when changes are made at the bottom of the tree structure. (Frakes and Gandel, 1989: 305)

The Library of Congress (LC) classification system is an enumerative scheme. It consists of 21 major classes. LC has separate schedules for each class. Each of these indexes are developed and published separately. Most research libraries in the United States organize their collections according to LC. However, LC does not lend itself to use in automated retrieval systems due the lack of a predictable basis for subject analysis and the lack of a logical hierarchy. (Ruble, 1987:57-60)

"To overcome the rigidity of an enumerated classification system, a more flexible type of classed system — known as a faceted classification — evolved. The theory

of faceted classification was developed by Raganathan, an Indian mathematician and librarian" (Frakes and Gandel, 1989:305).

Raganathan's Colon Classification (CC) was the first completely faceted library classification scheme. It consists of five facets which are related to each other in a fixed citation order. The five facets, in citation order, are Personality, Matter, Energy, Space, and Time (PMEST). Each of these facets consists of elemental terms. For example, Time may be subdivided as year, season, month, week, day, hour, etc. A subject then is composed of elemental terms from each of the facets listed in citation order. (Frakes and Gandel, 1989:305-306; Ruble, 1987:62-64; Williams, 1965:123)

> A major advantage of a faceted classification over a hierarchical scheme is the freedom it gives the indexer to synthesize terms to express complex concepts. All concepts do not have to be pre-determined at the time of creating the classification system. Rather, the indexer can synthesize terms from facets to create concepts as needed. A faceted scheme is also easier to update and modify since you can change one facet without affecting any others. (Frakes and Gandel, 1989:306)

Early work in classifying software applied enumerative approaches such as the ACM classification scheme (Wood and Sommerville, 1988:199). Prieto-Diaz first proposed use of a faceted classification scheme for software in his PhD dissertation (Prieto-Diaz, 1985; Prieto-Diaz and Freeman, 1987). His scheme consists of six facets: Function, Objects, Medium, System type, Functional area, and Setting. Ruble expanded upon Prieto-Diaz's work by implementing a faceted classification scheme and associated retrieval mechanism for software components (Ruble, 1987). Ruble's scheme consists of 11 facets: Form, Activity, Focus, Location, Language, Algorithm, Operating system, Hardware, Performance rate, Memory requirements, and Precision. Faceted classifications have been shown to be easily expandable and domain tailorable.

Keyword Systems. Perhaps the simplest method for representing reusable software components is through the use of keywords. For example, the REUsing Software Efficiently (REUSE) system is a menu-driven information retrieval system which classifies each component into one of four classifications: template, module, package, and program (Arnold and Stepoway, 1987). A menu of keywords is used to add new components and search for existing components. The list of keywords may change as obsolete components are deleted and new components are added.

A thesaurus relates synonymous terms. A thesaurus, in the context of information retrieval (IR) systems, is a controlled keyword list for describing the variety of relationships between vocabulary terms by means of a series of alphabetical entries, each of which is composed of a subject term, related terms, and associated role indicators. In this context, a thesaurus is not simply a list of synonyms. Rather, as information retrieval specialists adopted this term, they extended its meaning to include both an alphabetical list of allowed terms and the semantic relationships between the allowed terms. The semantic relationships between terms are identified by role indicators such as:

- UF — use for
- RT — related term
- NT — narrower term
- BT — broader term

(Frakes and Gandel, 1989:307)

The REUSE system also provides for free-form information to be stored with each software component. The REUSE system constructs and submits queries to the underlying information retrieval system based on user inputs. (Arnold and Stepoway, 1987)

A similar scheme is CATALOG, a

> high performance information retrieval system designed to allow end users to create, maintain, and search databases containing both formatted records, such as are typically found in DBMS, and unformatted records, such as text, which most DBMS handle poorly. ...
>
> CATALOG features a database generator which assists users in setting up databases, an interactive tool for creating, modifying, adding, and deleting records, and a search interface with a menu driven mode for novice users, and a command driven mode for expert users. The search interface allows full boolean combinations of search terms and sets of retrieved records, and sophisticated partial term matching techniques such as automatic stemming, and phonetic matching. (Frakes and Nejmeh, 1987:381)

Frakes and Nejmeh "built a small database of software modules using CATALOG. ...These modules were from SUPER, a system built at Bell Laboratories for interactive reliability analysis. The information used to index these modules was taken from the descriptive headers required of each module in the SUPER system" (Frakes and Nejmeh, 1987:381).

2.2.1.2 Uncontrolled Vocabularies.

> In an uncontrolled vocabulary, no restriction is placed on what terms can be used to describe an item. Uncontrolled vocabulary terms can be drawn from any source, but are usually drawn from the indexed objects themselves. Some potential advantages of using an uncontrolled vocabulary are:
>
> - Cost — Since the index terms are often drawn from the text of the indexed objects, the indexing task can be highly automated. This is usually much cheaper than human indexing.
> - Specificity – Since terms are unrestricted, indexing terms can be made as specific as possible.
>
> (Frakes and Gandel, 1989:308)

The Reusable Software Library (RSL) uses free text indexing for Ada parts. The functions, procedures, packages, and programs are indexed by free text terms assigned by the programmer. These terms supplement a hierarchical classification. (Burton and others, 1987)

2.2.2 Formal Specifications. Another suggested approach is to create formal specifications for the desired components and then attempt to find a matching specification in the library. Problems here include dealing with syntactical differences in semantically similar specifications (do they describe the same component?) and determining close matches in two formal specifications (Wood and Sommerville, 1988:200).

Litvintchouk and Matsumoto specify components using Clear, an algebraic specification language based on formal algebra. Their method applies *category theory*, the branch of mathematics dealing with properties characterizing classes of algebraic structures, to expressing the static semantics of systems. A category is a class of objects together with a set of mappings (morphisms) defined for each pair of objects in the category. The standard tool for reasoning about a category is a diagram which is a directed graph whose nodes correspond to objects in the category and whose edges correspond to morphisms in the category. Functors map objects and morphisms in one category to objects and morphisms in another such that morphism source, target, and the operations of identity and composition are preserved. A component is defined by a *theory* which is a finite set of sorts (data type identifiers) and operation symbols together with a finite set of axioms. Reuse is accomplished through theory-building operations such as combine, enrich, and derive. (Litvintchouk and Matsumoto, 1984)

2.2.3 Knowledge Engineering. Knowledge engineering creates a knowledge representation using artificial intelligence (AI) techniques.

> A central problem of IR has been how to represent the meaning of text or other records in a way comprehensible to a computer. The knowledge representation techniques used in AI systems offer promise in this direction. (Frakes and Nejmeh, 1987:383)

Knowledge engineering representation methods are typically divided into three classes: semantic nets, production rules, and frames.

> [A] semantic net approach [has been used for] ...document representation, production rules have been used to create an intelligent thesaurus, and natural language systems have been used to extract and formalize the information in medical documents (Frakes and Nejmeh, 1987:383).

2.2.3.1 Semantic Nets. "A semantic net is a directed graph whose nodes correspond to conceptual objects and whose arcs correspond to relationships between those objects" (Frakes and Gandel, 1989:309). Figure 3 shows a simple example of a semantic net representing sorting programs.

Semantic net representations suffer a serious practical problem. The processing required to determine whether a given semantic net matches, or partially matches, a library component representation can be extremely expensive. The comparison problems are instances of the graph and subgraph isomorphism problems. The subgraph isomorphism problem has been shown to be NP-complete while the graph isomorphism problem appears to be NP-complete but is still open (Garey and Johnson, 1979:202; Booch, 1987:342). Thus the time to perform the comparison, even using the best known algorithms, increases at least exponentially with respect to the number of nodes in the networks.

"No reuse systems based on semantic nets are described in the literature" (Frakes and Gandel, 1989:309).

2.2.3.2 Production Rules. Production rules follow an "IF condition THEN" structure where one or more attributes are tested against given values.

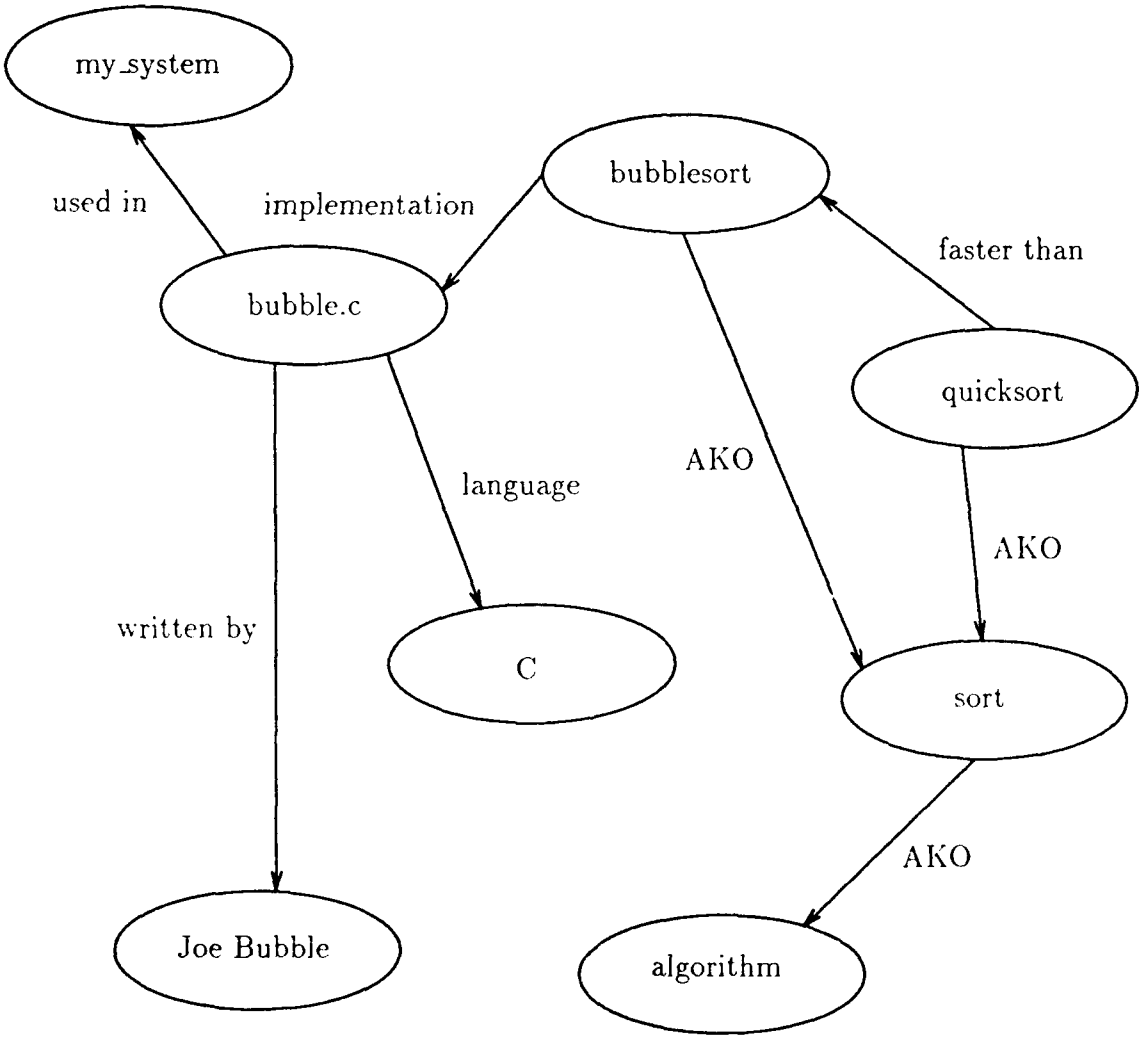


Figure 3. Semantic Net Example (Frakes and Gandel, 1989:309)

These production rules are typically stored in expert system knowledge bases. When rule-based systems are used as a representation method, each software component must be described by the attributes represented in the rule base. Several projects have successfully employed rule-based expert systems for software reuse (Frakes and Gandel, 1989:309–310).

One such system is the Modeling Expert System (MES), developed at AT&T Bell Laboratories, which aids in generating models for new transmission equipment by reusing models of existing equipment. The models are written in PL/I. The MES itself is written in OPS5 and Franz LISP. Selected attributes of the desired model fire the appropriate MES production rules which select existing models to be used as base models. The selected base model is then modified with the help of scripts written in the awk pattern-matching language under the UNIX operating system. (Rosales and Mehrotra, 1988)

2.2.3.3 Frames. A frame is a data structure consisting of slots and fillers, where a slot is a location for a particular attribute and a filler is the value of that attribute. A type of inferencing known as inheritance is usually used to access the knowledge contained in frames. An "is a" (ISA) relationship between two frames logically generates a resulting frame containing the slots and fillers of the related frames. For example, a frame named CAR would encode certain information common to all cars. Another frame named VW would have different slots and fillers unique to Volkswagens. VW ISA CAR means that the VW frame inherits the knowledge encoded in the CAR frame.

Several reuse projects have successfully used frames as their representational method. One notable example is work done as part of the Eclipse project which is now being continued in an ESPRIT-funded reuse project. ESPRIT is the European Strategic Program for Research in Information Technology administered by the Commission of the European Communities (Gibbs and others, 1987:4). The developers

of the Eclipse system used component descriptor frames which encode the semantics of language as well as the syntax thus allowing retrieval of components using a more natural language than previous retrieval languages. (Wood and Sommerville, 1988)

Another example is LaSSIE, "a prototype tool ...that uses a frame-based description language and classification inferences to facilitate a programmer's discovery of the structure of a complex system. It also supports the retrieval of software for possible reuse in a new development task" (Devanbu and others, 1989:110). The LaSSIE knowledge base has four object types: OBJECT, ACTION, DOER, and STATE. The relationships are captured by slot-filler relationships between OBJECTs, ACTIONs, and DOERs (Devanbu and others, 1989:111). Classification is accomplished by building "explicit descriptions of the actions performed by different parts of the system, then [using] formal inference to build a taxonomic hierarchy, where all IS-A links are derived from the descriptions themselves" (Devanbu and others, 1989:112).

2.2.4 Hypermedia. Ghisio and associates "focussed on techniques based on the information structure" (Ghisio and others, 1987:386). They identified

> two useful techniques for structuring information:
>
> - structuring through explanation levels;
> - structuring through partial views.
>
> The structuring through levels of explanation enables the user to read only the descriptions he is interested in. The information is organized in one or more levels, in a hierarchical way: the first level will contain the main description while the other levels contain references describing aspects which could need further explanations.
>
> The structuring through partial views enables the user to gather different chunks of information related to the same topic. This technique is based on the concept that the information could be decomposed in several information chunks; each chunk could then be linked to related chunks through explicit connections. (Ghisio and others, 1987:386–387)

"A hypertext is a structure that allows a user to move from place to place in a body of text via links" (Frakes and Gandel, 1989:311). A hypertext system can be used to organize knowledge as a graph structure of frames. Such a "knowledge graph" may be entered at different points and traversed in different ways depending upon the experience of the user. Thus, hypertext can be viewed as a paradigm of knowledge engineering which imposes a modular, interactive discipline on both creators and users.

In working with knowledge graphs it is conceptually useful to define individual frames as objects of an abstract data type which may have some components and operations for manipulating each of those components. Knowledge graphs in general, and hypertexts in particular, have a domain-independent interconnection structure and set of operations that facilitates several modes of graph traversal such as browsing, retrieval, and reference. Such a domain-independent set of operations provides the means to operate on graphs and frames, such as navigating the graph structure, independently of the knowledge domain being considered. Each node (frame) has a domain-dependent internal structure and may provide domain-dependent operations or tools which know about, and can manipulate, objects of the particular knowledge domains. (Wegner, 1983:41)

Hypermedia, an extension of hypertext, enables the same nonlinear movement through other forms of information in addition to text. Hypermedia systems give the user a tool to create smoothly integrated webs of information. Such an information web consists of nodes, or elements, annotated with *links* which point to other nodes in the web. Such link annotations may be placed anywhere in a node's contents and can be made visible or invisible. Hypermedia systems also provide browsing tools for navigating the structure of the information web and viewing the contents of the nodes. (Biggerstaff, 1987:1-2; Biggerstaff and Richter, 1987:43)

> The single most important property provided ... is the connectivity provided through the annotation system. This translates into an important change in the way information is retrieved and thereby, into a fundamentally new way for the Software Engineer to operate with the design model of the target system that he or she is intending to reuse. (Biggerstaff, 1987:6)

This concept has been used as a representation method for reusable software components. Hypertext and hypermedia systems aid the user in understanding a component by providing instant access to supporting information within a few keystrokes or mouse button clicks. In addition to existing code, such supporting information may include text descriptions, graphical diagrams, explanations of design decisions, and other design information. (Biggerstaff and Richter, 1987: 43–44). Unlike other representations, hypermedia allows the user to be intimately involved in the search for components. Thus, search is just part of the pattern of the user interacting with the model of the component to be reused, navigating his or her way through a complex space of information describing the component. (Biggerstaff, 1987:1)

The Student Engineering Environment for Reusable Software (Seer) at the University of Maine (Latour and Johnson, 1988) uses a hypermedia-like approach to representing the Booch components (Booch, 1987). The Seer user interface is a windowing system with four types of windows:

- module class window: displays a button for each of the top-level classes in the Booch taxonomy.
- class state window: displays a tree of buttons with the root of the tree being the currently selected module class and the branches of the tree being the factors in the Booch classification hierarchy subordinate to the selected module class. The buttons representing the currently selected factors are highlighted on the screen.
- information web window: displays a graphical web of information buttons about the currently selected module class and factors.

- text/graphics window: displays the textual and graphical information associated with the selected information web button.

> The information ...differs in a small but critical way from the same information presented in a printed document. The user has control over the order in which he or she investigates that information. The user can broadly sample each entity in the diagram first before investigating any one entity in depth, or he or she can dive deeply into the details of some entity. On such a depth first investigation, the user may come to a point where references to unexplored branches of information will redirect him or her back up the information graph. Hypermedia makes such redirection easier than bouncing around in a linear document. It does not however, relieve the developer of the responsibility to carefully and thoughtfully organize the information. Hypermedia systems are helpful, but not magic. (Biggerstaff, 1987:4)

2.3 Component Collections

Any discussion of reusable software component representation methods must consider the components to be represented. Numerous collections of reusable software components exist.

2.3.1 Ada components. Ada, the standard high-order programming language for the DOD, is a modern programming language with several characteristics specifically designed to support development of large systems composed of reusable software components. For example, Ada has a wide variety of program units (procedures, packages, and tasks) each of which has a specification part that defines the interfaces available for interconnecting program units in composite systems and a body which contains the implementation details of the program unit. Thus, Ada supports the modern software engineering concept of information hiding. Ada's strong typing rules constrain module interconnections and allow compile-time consistency checking between the formal parameters defined in the specification and the actual parameters established when the module is invoked. Parameterized templates, known as generic

program units, allow the common features of families of software components to be captured by a single generic definition. Finally, Ada's program libraries and separate compilation capability encourage reuse of program units. (Wegner, 1983:31)

Common collections of components written in Ada include:

- the Booch components, developed by Grady Booch (Booch, 1987) and distributed by Wizard Software. This collection consists of 501 components implementing frequently needed structures and tools in a predominantly object-oriented design approach.
- the Generic Reusable Ada Components for Engineering (GRACE) components, distributed by EVB Software Engineering. These components are classified using a modification of the taxonomy Booch developed for his components.
- the Common Ada Missile Packages (CAMP), developed by McDonnell Douglas Astronautics Company under sponsorship of the Air Force Armament Laboratory. These packages are the result of an analysis of the tactical air-launched missile domain.
- the Ada Software Repository (ASR), available through the SIMTEL20 computer at White Sands Missile Range. This repository is a diverse collection of systems, subsystems, and packages.

All of the Ada components mentioned above are available in both source and object code form.

2.3.2 Non-Ada Components. Common collections of software code components written in languages other than Ada include:

- The UNIX components consisting of utilities, pipes, and filters. These components are written in the C programming language and are usually provided in both source and object code forms on UNIX systems.

- The IMSL libraries of FORTRAN mathematical and statistical routines. These routines are usually available only in object code form.

III. Approach

The first section of this chapter describes the process used to select the component collection for this study, the characteristics of the component collection, and the classification scheme used to represent the components. The second section of this chapter describes the hypertext approach used to implement the Reusable Software Component Representation System (RSCRS).

3.1 The Component Collection

As mentioned in Section 1.2, little is known about reuse other than code. Although reuse at higher levels (e.g., analysis, specification, or design) may, in the long term, offer more significant benefits, reuse of code components appears to offer the best near term advantages. Thus, this study concentrated on the representation of reusable code components. Early in this research, a decision was made to utilize an existing collection of reusable software components rather than to develop a new collection. This decision was based on the fact that most existing collections contain a larger number and wider variety of tested components than could be developed within the available research time. In addition, it is reasonable to assume that an existing collection would not have been specifically developed to easily fit into the selected representation methods. Also, the focus of this research was on component representation rather than component creation. Use of self-developed components could have introduced author bias. Finally, most reusable software components in production libraries will be represented in the system by someone other than the author of the component, using representation methods which may not have been anticipated by the author of the component during component development.

3.1.1 Selection Criteria. Several criteria were used to select the component collection to be represented. First, to make this work widely applicable, the com-

ponent collection had to be generally applicable across domains. Second, the component collection had to be readily available in source code form. Availability of source code should encourage reuse because a similar existing component could then be modified if the exact component desired was not available. A third desirable criteria was use of the Ada programming language due to its status as the standard high-order programming language within the Department of Defense and its increasing use in other government agencies and commercial enterprises both in the United States and abroad. Fourth, an object-oriented component collection was desired as "...object-oriented design is the most promising technique now known for attaining the goals of extendability and reusability" (Meyer, 1987:51). Fifth, the components had to be already tested. Reuse of untested components introduces considerable risk into a software development effort, perhaps even enough to outweigh the potential benefits of reuse.

3.1.2 Selection Results. The Booch components, a collection of 501 reusable Ada components created by Grady Booch and distributed by Wizard Software, were selected for this research effort. This component collection met all of the selection criteria outlined above. In addition, the components were specifically designed to be reusable. Most of them are parameterized components implementing frequently required structures and algorithms. Finally, Booch has written a textbook (Booch, 1987) which documents the design considerations of the components in some depth.

Other candidate component collections, mentioned in Chapter II, were considered. They were rejected because each failed to meet at least one of the criteria established to select the component set. For example: the GRACE components were rejected because they were not available at this institution, the CAMP components were rejected because they are primarily applicable to a very specific problem domain, the ASR components were rejected because there is no verification testing performed on code submitted to the repository, the UNIX components were rejected

because they are not coded in Ada, and the IMSL components were rejected because they are not coded in Ada and are not usually available as source code.

3.1.3 Representation of Components. The purpose of a software component representation is to provide information to locate desired components and assess their suitability for reuse in a given application without having to examine the code itself. A hybrid classification scheme, textual descriptions, abstract data type (ADT) descriptions, and time and space complexity information were used to represent the components. The classification scheme provides a means for locating potentially useful components. Textual descriptions provide easily read definitions of terms and descriptions of components. ADT descriptions provide a clear understanding of the operations exported by the components. Results of time and space complexity analyses provide information about the components' requirements for computer processing and storage resources. Thus, the RSCRS representation satisfies the stated purpose of a software component representation.

Obviously, other representations could have been implemented. For example, a different classification scheme could have been chosen or a rule- or frame-based knowledge system could have been implemented. The chosen representation was selected because of its compatibility with the hypermedia implementation discussed in Section 3.2.

3.1.3.1 Classification Scheme. The representation method selected for implementation was Booch's classification scheme (Booch, 1987). This classification scheme conveniently matches the component collection. It appears to be reasonably extensible to other components. This classification scheme is a hybrid which combines enumerated and faceted approaches. The enumerated portion of the classification scheme is a hierarchy that divides the component collection into successively narrower classes. Figure 4 illustrates the hierarchical portion of the classification.

- Reusable Software Component
  - Structure
    - Monolithic
      - Ordered
        - Stack
        - String
        - Queue
        - Deque
        - Ring
      - Unordered
        - Map
        - Set
        - Bag
    - Polylithic
      - List
      - Tree
      - Graph
  - Tool
    - Utility
      - Primitive
        - Character
        - String
        - Numeric
          - Integer
          - Floating Point
          - Fixed Point
        - Calendar
      - Structure
        - List
        - Tree
        - Graph
      - Resource
        - Storage Manager
        - Semaphore
        - Monitor
    - Filter
      - Input
      - Process
        - Translate
        - Expand
        - Compress

- Reusable Software Component (Continued)
  - Tool (Continued)
    - Filter (Continued)
      - Output
    - Pipe
    - Sorting
      - Total Ordering
        - Internal
          - Insertion
            - Straight
            - Binary
            - Shell
          - Exchange
            - Bubble
            - Shaker
            - Quick
            - Radix
          - Selection
            - Straight
            - Heap
        - External
          - Natural Merge
          - Polyphase
      - Partial Ordering
    - Searching
      - Primitive
        - Sequential
        - Ordered Sequential
        - Binary
      - Structure
        - List
        - Tree
        - Graph
    - Pattern Matching
      - Simple
      - Fast
        - Knuth-Morris-Pratt
        - Boyer-Moore
      - Regular Expression

Figure 4. Booch Classification Structure

Each of the lowest levels in the classification structure represents a class of components.

The components in each of the classes are distinguished by their *forms*. The forms represent the time and space properties of a component. Booch defines eleven categories of forms which are identified in Figure 5. Not all categories of forms are

Concurrency	Space	Garbage Collection	Iterator Export
Sequential	Bounded	Managed	Iterator
Guarded	Unbounded	Unmanaged	Noniterator
Concurrent		Controlled	
Multiple			
Balking	Priority	Object Domain	Cacheing
Balking	Priority	Simple	Cached
Nonbalking	Nonpriority	Discrete	Noncached
	Link	Degree	Directedness
	Single	Binary	Directed
	Double	Arbitrary	Undirected

Figure 5. Forms of Booch Components

applicable to all components. It is convenient to view these categories of forms as a faceted portion of the classification system because the forms do not have any type of hierarchical relationships to each other. The forms, instead, provide faceted views of the components.

The classification presented in Figures 4 and 5 is a superset of the taxonomy Booch uses to name the components. His component naming taxonomy typically, especially for the structure components, consists of the component class (a leaf node of the classification tree) and the applicable forms for the component (for example, Stack_Sequential_Bounded_Managed_Iterator). His naming conventions for tools are not quite so orderly, but the tool name always includes any applicable forms (for example, Topological_Sort_Bounded_Managed).

3.1.3.2 Textual Descriptions. Since many of the terms in the classification scheme are somewhat specialized or uniquely defined, each term is defined and explained within the RSCRS. These explanations are extracted from (Booch, 1987).

(Booch, 1987) is tutorial in nature. It provides the rationale for Booch's approach to reusable software component design and implementation. He views each component as implementing some real-world abstraction. Thus, the text provides an explanation of the mathematical structure of each abstraction, the outside and inside views of the abstraction, and the design considerations from the perspectives of utility and efficiency.

The information presented in RSCRS concentrates on the abstract properties of the components, ignoring the details of how the components operate except when understanding the inside view is necessary for proper use of the component.

3.1.3.3 Abstract Data Types.

> An abstract data type describes a class of objects through the external properties of these objects instead of their computer representation. More precisely, an abstract data type is a class of objects characterized by the operations available on them and the abstract properties of these operations.
>
> It turns out that abstract data types, which provide an excellent basis for software specification, are also useful at the design and implementation stage. In fact, they are essential to the object-oriented approach, and enable us to refine the definition of object-oriented design: Object-oriented design is the construction of software systems as structured collections of abstract data-type implementations. (Meyer, 1987:53)

Abstract data type (ADT) descriptions are provided in RSCRS for the structure components. ADTs are not applicable to the tool components because the tool components are not objects. Each ADT consists of the name of the structure class, a short definition of the structure, the suffered operations provided by the class,

and the applicable forms. The operations are categorized as constructors, selectors, iterators, and exceptions.

3.1.3.4 Time and Space Complexity. Time and space complexity may be crucial in selecting the proper component when processing power or storage capacity is at a premium or when the software system performance requirements are tight. Thus, the RSCRS includes the results of this type of "order-of" analysis. The time and space complexity analysis results presented in the RSCRS were extracted from (Booch, 1987).

3.2 The Hypermedia Implementation

As implied in Chapter II, hypermedia can be viewed as a means to present underlying representation methods. For example, some frames in a hypermedia environment could contain a semantic net representation while other frames could contain a faceted classification while yet other frames could allow on-line browsing of the source code for the component. Thus, hypermedia may provide the capability to integrate multiple representations in a single presentation. Of course, each additional representation method implemented for a component collection adds to the development and maintenance costs of the collection.

The decision to implement RSCRS as a hypermedia-based system left two alternatives for hardware systems to serve as host platforms. These alternatives were personal computers (PCs) or the school's network of Sun workstations. The Sun network was chosen for three reasons. First, not many PCs have enough disk storage to hold the complete collection of Booch components in addition to their existing requirements for mass storage space. Thus, stand-alone PCs are not good candidates for hosting a library of reusable software components. Second, an Ada compilation system is projected for installation on the Sun network shortly after this thesis effort is completed. A means to locate components for Ada software

developments would, thus, be a useful addition to the network. Third, the only means currently available at this school to locate these components is a simple, text-oriented capability on one of the school's central computer systems.

The Knowledge Management System (KMS) by Knowledge Systems, a hypermedia system supporting text and graphics presentation, was selected to serve as the principal presentation vehicle for the prototype Reusable Software Component Representation System (RSCRS). KMS allows the user to organize knowledge in chunks called frames. Frames are grouped together into framesets. A frameset will usually consist of frames containing related information. Appendix A describes the features of KMS.

KMS was selected by default as it is the only hypermedia system installed on the selected computer system. RSCRS was the first sizable system implemented in KMS at this institution.

The following subsections describe how the representation information is presented within the hypermedia environment of RSCRS. Figure 6 illustrates the organization of the RSCRS representation. Appendix B presents a number of sample KMS frames from RSCRS.

3.2.1 Classification. A frameset named BT, an acronym for Booch Taxonomy, was created in which the first frame contains an item for each of the choices at the highest level in the classification hierarchy (i.e., "Structures" and "Tools"). Subsequent frames within the BT frameset contain items for the next lower level in the classification hierarchy, based upon the item selected in the parent frame. Thus, a RSCRS user selecting the "Tools" item in the first frame is presented with a frame containing the items "Utilities", "Filters", "Pipes", "Sorting", "Searching", and "Pattern Matching".

The implementation of the faceted part of the classification system proved somewhat challenging. Initially, a single frame, enumerating all of the component

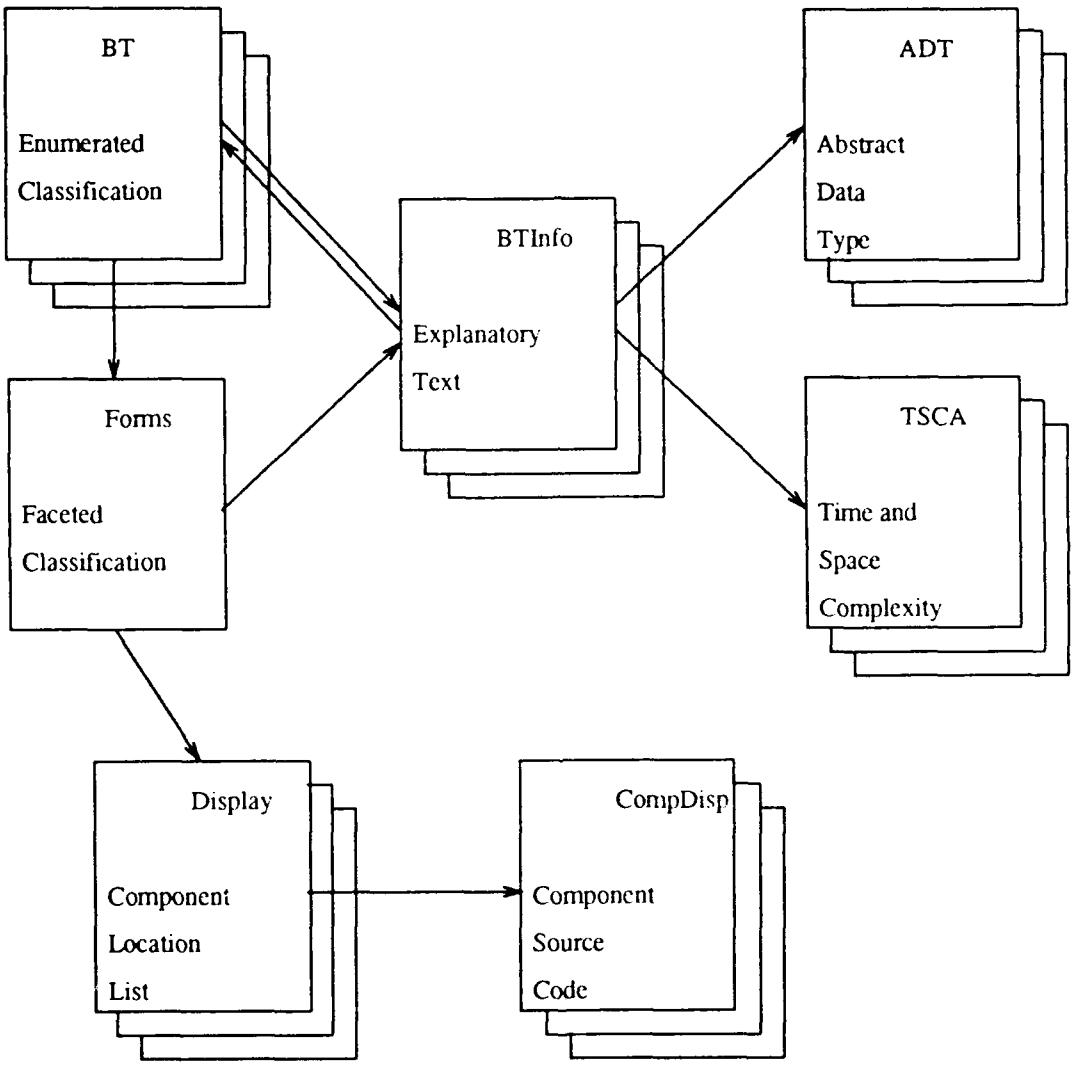


Figure 6. RSCRS Organization

forms without regard to the class of component actually being investigated, was implemented. However, attempts to use that frame to locate the desired components resulted in an unacceptable number of failed searches due to the fact that only certain categories of forms are applicable to any given class of components. Any attempt to locate a component by specifying an inappropriate form results in a failed search.

To remedy this problem, a KMS Action Language program was developed to dynamically create a frame containing items for only those forms applicable to the component class being investigated by the RSCRS user. This approach resulted in almost no failed searches. However, because the KMS Action Language is interpreted rather than compiled, a short, but noticeable, delay occurs while the forms frame is being created.

An alternative approach would have been to create a separate forms frame for each component class. This approach would have eliminated the delay due to creating the frame dynamically. In retrospect, this approach would also have required fewer frames to implement.

3.2.2 Explanatory Text. Within the prototype representation system, an annotation link is associated with each term in the Booch classification. Selecting an annotation link leads to an information frame containing text extracted from (Booch, 1987) which defines the term and provides related information. Within an information frame, references to other terms are also provided annotation links to the information frames for the other terms. Thus, a web of information is established. Figure 7 provides a simple example of a hypothetical information web.

Navigation through the web is very simple. It is usually quite easy to backtrack to a previously visited frame because KMS retains the sequence of frame traversals. All of these information frames are contained in a frameset named BTInfo.

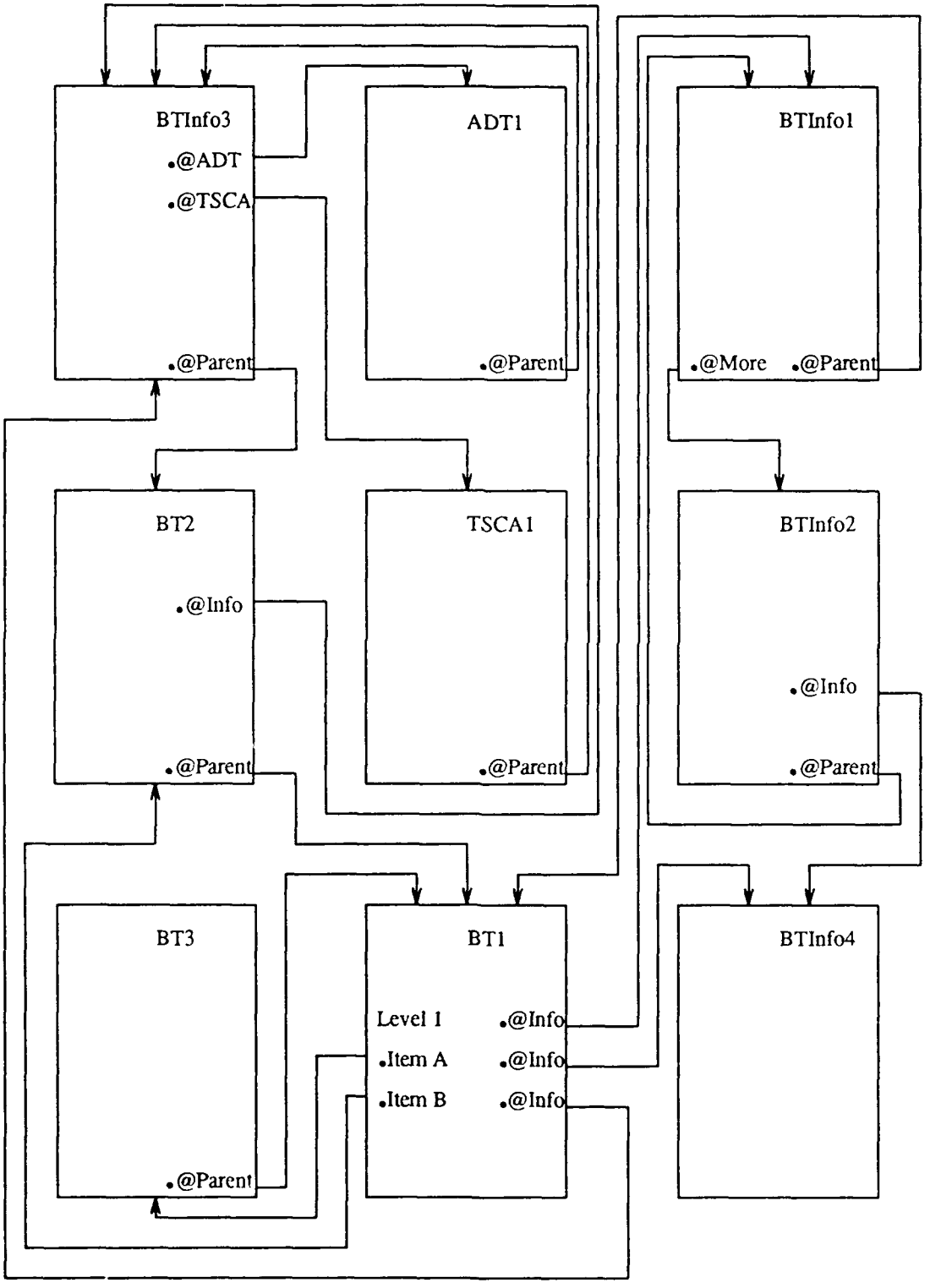


Figure 7. Information Web Example

3.2.3 Abstract Data Types. The first information frame for each class of structure components contains a link to an abstract data type description of the component class. All of the ADT descriptions for the structure components are contained in a frameset named ADT. The tool component classes do not have abstract data type descriptions associated with them as they are not objects in the strict sense.

3.2.4 Time and Space Complexity. The first information frame for each class of structure components also contains a link to a table presenting the results of an analysis of the time and space complexity of the component operations. All of the time and space complexity analysis tables are contained in a frameset named TSCA. For the tool components, the time and space complexity information is integrated into the text of the components' information frames consistent with the approach in (Booch, 1987). The time and space complexity information for the tool components could have been put into separate tables to provide consistency in representation throughout the system.

3.2.5 Component Locator. Booch's naming taxonomy is the basis for a primitive component locating capability within RSCRS. The source code for each Booch component is contained in two separate files, having rather cryptic UNIX file names. A set of index files — one file per component class — was created which associates the actual component name with its UNIX file name. The selected component class and forms are used by a KMS Action Language program to generate a UNIX awk program which examines the appropriate index file to locate the file name of the desired component(s). After building the awk program file, the KMS Action Language program issues ("shells") the awk program to the operating system to be run. Upon completion of the awk program, the KMS Action Language program continues, building a frameset, named Display, which contains the component name, specification file name, and body file name for each selected component.

Like KMS Action Language programs, awk programs are interpreted not compiled. Thus, the combination of the action language program, the shell command, and the awk program result in an sometimes uncomfortable delay from the time the user selects the "Locate Component(s)" item on the forms frame until the results are displayed. The desire to reduce this delay was the factor which drove the use of several index files instead of a single index file. Breaking the single index file into smaller index files resulted in a much smaller search space to be examined by the awk program.

Table 3 presents the response times (rounded to the nearest second) observed between selection of the "Locate Component(s)" item on the Forms frame and the subsequent display of the first Display frame. The response times in the table are those observed, during a period of typical system load, for a sample of ten observations for randomly selected components. The sample mean response time for the Locate function is 10.1 seconds with a sample standard deviation of 1.287 seconds.

Component Name	Response Time (Seconds)
Stack_Multiple_Unbounded_Managed_Noniterator	12
String_Guarded_Unbounded_Unmanaged_Iterator	11
String_Guarded_Bounded_Managed_Iterator	9
Queue_Nonpriority_Nonbalking_Sequential_Unbounded_Managed_Iterator	11
Deque_Nonpriority_Balking_Multiple_Unbounded_Unmanaged_Iterator	11
Map_Simple_Noncached_Guarded_Unbounded_Unmanaged_Noniterator	11
Map_Simple_Cached_Multiple_Bounded_Managed_Noniterator	9
Bag_Simple_Sequential_Unbounded_Unmanaged_Noniterator	10
Bag_Simple_Sequential_Bounded_Managed_Noniterator	8
Output_Filter	9

Table 3. Typical "Locate Component(s)" Response Times

3.2.6 Browsing. The component classification, definition, design information, abstract data type description, and time and space behavior provide much

of the information necessary to determine the suitability of a component for reuse in a particular application. However, a complete understanding of the component can only be achieved by also inspecting the actual source code. Thus, upon locating a candidate component, the RSCRS user is given the opportunity to browse the source code for the specification and/or body of the component. This capability is especially important when the selected component does not appear to completely satisfy the user's requirements. Browsing can provide the knowledgeable user a quick way to identify the scope of possible changes required to adapt the component to the user's needs.

As shown in Table 4 and Figure 8, using a set of ten randomly selected components, the larger the component source code file is, the longer RSCRS requires to bring the file into a KMS frameset (named CompDisp) for browsing.

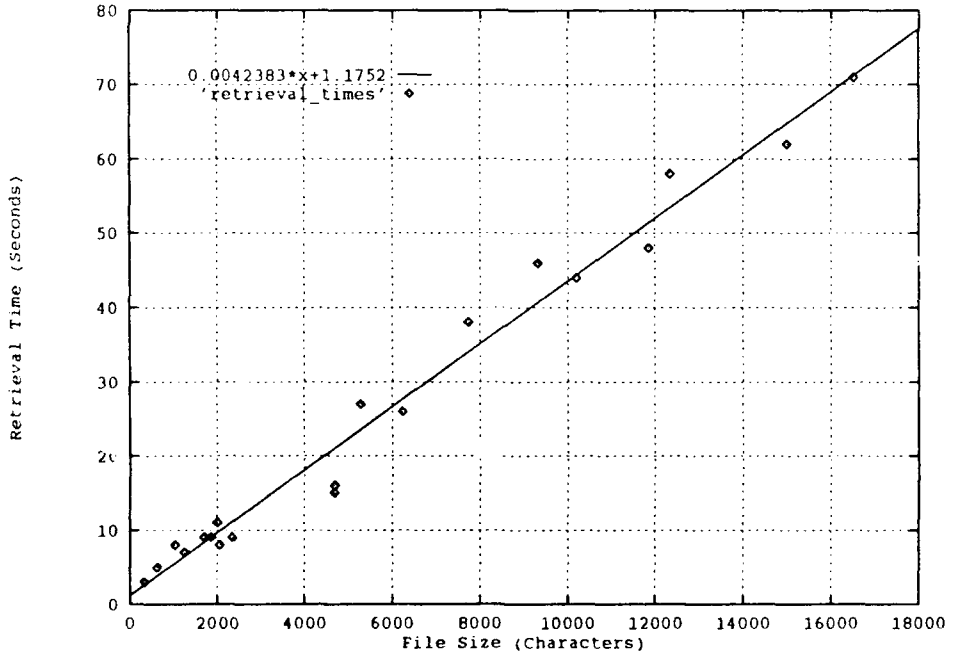


Figure 8. Component Retrieval Time vs. Component Size

For the very small component specification files this delay is not at all unacceptable, but the delay can be quite annoying when a large file is retrieved. Again, this is primarily a result of the fact that the KMS Action Language is interpreted rather than compiled.

Component Name	Component Size (Characters)	Retrieval Time (Seconds)
Stack_Multiple_Unbounded_Managed_Noniterator (Spec)	1033	8
Stack_Multiple_Unbounded_Managed_Noniterator (Body)	6249	26
String_Guarded_Unbounded_Unmanged_Iterator (Spec)	4687	15
String_Guarded_Unbounded_Unmanged_Iterator (Body)	16525	71
String_Guarded_Bounded_Managed_Iterator (Spec)	4698	16
String_Guarded_Bounded_Managed_Iterator (Body)	15012	62
Queue_Nonpriority_Nonbalking_Sequential_Unbounded_Managed_Iterator (Spec)	1247	7
Queue_Nonpriority_Nonbalking_Sequential_Unbounded_Managed_Iterator (Body)	5283	27
Deque_Nonpriority_Balking_Multiple_Unbounded_Unmanaged_Iterator (Spec)	1866	9
Deque_Nonpriority_Balking_Multiple_Unbounded_Unmanaged_Iterator (Body)	9327	46
Map_Simple_Noncached_Guarded_Unbounded_Unmanaged_Noniterator (Spec)	1702	9
Map_Simple_Noncached_Guarded_Unbounded_Unmanaged_Noniterator (Body)	7748	38
Map_Simple_Cached_Multiple_Bounded_Managed_Noniterator (Spec)	2002	11
Map_Simple_Cached_Multiple_Bounded_Managed_Noniterator (Body)	10204	44
Bag_Simple_Sequential_Unbounded_Unmanaged_Noniterator (Spec)	2050	8
Bag_Simple_Sequential_Unbounded_Unmanaged_Noniterator (Body)	12336	58
Bag_Simple_Sequential_Bounded_Managed_Noniterator (Spec)	2344	9
Bag_Simple_Sequential_Bounded_Managed_Noniterator (Body)	11860	48
Output_Filter (Spec)	341	3
Output_Filter (Body)	631	5

Table 4. Typical Component Retrieval Times

3.3 Summary

The RSCRS consolidates a large amount of diverse information about the component collection in a system which enables the non-linear acquisition of the information. Unlike a book, which presents the same information in a linear manner, the hypermedia-based representation allows users to choose the path or paths they

desire to obtain the information without being concerned about losing their place in a chain of cross-references. This non-linearity provides a synergistic effect which is not easy to obtain when examining the same information presented in linear text form.

IV. Summary, Conclusions, and Future Work

4.1 Summary

Many methods have been proposed to represent reusable software components. The work on this thesis resulted in creation of a prototype representation system to serve as a testbed for further research on the subject. The prototype system, the Reusable Software Component Representation System (RSCRS), is a system which presents component representations within the framework of a hypermedia system. This system provides as a tool for the students and faculty at this institution to learn about the component collection, as well as locate and retrieve desired components.

Hypermedia, with its capability to provide multiple views into software components appears to offer much promise as a vehicle for representing reusable software components.

4.2 Conclusions

Creating a hypermedia representation for software components is a very labor-intensive, time-consuming effort. To represent the complete collection of 501 Booch components required over 320 man-hours of effort, not including the time required to become familiar with the KMS hypermedia system itself.

Booch's component naming taxonomy (see Section 3.1.3.1) is not very useful for representing software components. It is inconsistent in its conventions. On the other hand, the hybrid classification structure implied by the presentation in (Booch, 1987) proved quite amenable to implementation within a hypermedia environment. The hierarchical portion of the classification was implemented in a very straight-forward fashion because each branch in the classification hierarchy naturally corresponds to a link in the hypermedia web structure. The faceted portion of the classification was

not quite so straight-forward to implement due to the lack of structural relationships between the facets.

KMS as an implementation vehicle does not provide some of the functionality which would make the system truly useful, not only for component representation, but for any type of knowledge representation. The fact that links between frames are not automatically adjusted when a frame name or frameset name is changed or deleted proved very inconvenient when developing the RSCRS. As development progressed and the desire arose to split a frameset into two, all references to the old frame names for the frames which were moved had to be manually changed.

Although the prototype RSCRS offers a convenient method for locating desired Booch components, the delays while waiting for the locate and retrieval programs to complete their tasks can be somewhat annoying. The slow execution time for KMS Action Language programs is annoying, at best, and a disincentive to using the system. The lesson to be learned here is to understand, before beginning development, the demands which will be placed on the underlying hypermedia system and find a system which can meet those demands before proceeding with development.

4.3 Recommendations for Future Work

Because the task of actually representing the components is so time-consuming, an automated aid to component representation would help make a representation system more practical, by making it less manpower-intensive to create and maintain.

The explanatory text for an item in the current version of RSCRS is presented as a sequence of frames. The granularity of this information is too large. Having less information on each frame would result in a larger number of frames, but would permit more accurate cross referencing between information items.

Addition of a specialized query capability would be very useful to save the user from having to traverse the information network to find a specific piece of information. KMS does provide limited capability to search a frameset for a specified

term. However, what is really needed is a means for the user to formulate arbitrarily complex queries and have the system locate the desired information. A desirable feature of such a query engine would be the capability to handle synonymous terms.

A related feature which could be very useful useful would be to provide the capability to directly specify the naming taxonomy of a desired component. Such a feature would relieve the knowledgeable user from having to traverse the classification tree and forms frames in order to retrieve a known component.

Extracting the time and space complexity information for the tool components from the explanatory text frames and putting the information into separate tables would make the representation of structure and tool components more uniform. A similar argument could be made for the information concerning operations exported by the components. Although the tool components are not objects in the strict sense, many of them are crafted to export composite operations applicable to a particular class of structures. Thus, tables similar to the abstract data type tables for the structure components could be created for the tool components.

The current RSCRS suffers the practical limitation of permitting only a single user at a time to work in the Forms, Locate, and Display framesets. Adding the capability for multiple concurrent users is a firm requirement before the system could be declared ready for use in a real software production environment. One change which would contribute to permitting multiple users would be to create static frames to contain the forms for each component class instead of dynamically creating a single forms frame. This would also require fewer frames than the programs which generate the current forms frame.

Instead of storing the source code as separate files, it is possible to store the source code in KMS frames. This would eliminate the long delays the current system exhibits when retrieving the source code for a component. Since the source code would already be in frames, retrieval time would be virtually instantaneous. Using this approach, a component would have to be copied from its KMS frames into a

regular file in order to be used in a development effort. KMS provides the required operations to support such an approach and automating the process would not be difficult. Changing to this approach would also eliminate one of the areas of potential conflict in a multiple concurrent user scenario. However, this approach would mean that the only access to the components would be through KMS (using either RSCRS or the domain independent KMS operations.)

RSCRS provides a suitable platform for comparing representation methods. Several of the implementation problems one could expect to encounter in adding additional representations to RSCRS have already been encountered and solved in the current system. The next logical step is to implement additional representation methods within RSCRS and then to perform an empirical study comparing the various representation methods. This type of comparative study has not yet been published. Such a study would be a welcome contribution to the field of software component representation.

Appendix A. *Knowledge Management System Overview*

The Knowledge Management System (KMS) by Knowledge Systems is a distributed hypermedia system for text and graphics. It runs on Sun and Apollo workstations. Most of the information in this appendix, and more, may be found in the documentation provided with KMS (*Getting Started with KMS*, 1988; *Introduction to KMS*, 1988; *KMS Reference Manual*, 1988; *KMS Action Language Manual*, 1988).

KMS allows the user to organize knowledge in chunks called frames. Frames are grouped together into framesets. A frameset will usually consist of frames containing related information.

To enter KMS on a Sun workstation, the user types `kms` at the system prompt in a Sunview window. KMS will enlarge the Sunview window to occupy almost the full screen. This large Sunview window will contain three KMS windows: one small window across the top and two windows below. The top window is for messages from KMS and user input to KMS. Each of the lower windows displays a KMS frame with a standard set of command items across the bottom of the window. Initially, the left KMS window displays the user's home frame and the right KMS window displays the top level KMS information frame. The KMS information frames provide on-line access to all of the KMS documentation.

Items on a frame may be linked to other frames (not necessarily in the same frameset). The frames and their links thus form an information network. KMS provides a set of domain-independent operations for navigating through the network of frames and manipulating items, frames, and framesets. These operations include creating, clearing, and deleting frames; adding, modifying, and moving items on a frame; establishing and removing links between frames; and navigating through the network.

KMS remembers the links which have been traversed and provides a mechanism (the Back command item) for revisiting frames in the reverse order from that originally viewed. Occasionally, the user may want to go directly to a particular frame that is not linked to the current frame. To do this, the user may click on the Goto command item at the bottom of the frame and then type the name of the desired frame.

Because the network of hypertext frames may be arbitrarily complex, an inexperienced KMS user might worry about getting lost or disoriented. This is not nearly the problem it may initially appear to be as KMS provides a number of ways to get back to familiar territory in addition to the Back and Goto command items. The Home command item returns the user to the home frame. Additionally, the creator of a frame will often place an @Parent item in the lower right corner of the frame. Clicking on this item displays the logical parent frame (often the same frame that the Back command item leads to).

Each item in a frame has attributes associated with it. Two important attributes are Link and Actior The Link attribute designates the frame to display when the item is selected. The existence of a link for an item is indicated by a o preceding the item. The Action attribute designates a sequence of actions to be performed when the item is selected. The existence of an action for an item is indicated by a • preceding the item. KMS includes a general-purpose, block-structured programming language, the KMS Action Language, which gives access to all of the functions provided by the system. The KMS Action Language allows the author or user of an information web to automate sequences of actions. The KMS Action Language is somewhat primitive in terms of the control constructs it provides. For example, a single iteration construct (Repeat) is provided. There are no For, While, or Until constructs.

Another example of the primitive nature of the KMS Action language can be found in its conditional constructs. There are no If-Then-Else or Case constructs

available, only an `If`. Thus, to perform an `If-Then-Else` requires two `If` statements; the first to test for the condition, the second to test for the complement of the condition. Additionally, because there are no expressions in the language, only a single condition may be tested at a time. That is, compound conditions must be decomposed into a sequence of several `If` statements.

Except for a sequence of items in a frame, all transfers of control are represented by a text item linking to another frame. For example, to iterate a sequence of KMS Action Language statements, they must be placed in a separate frame. At the point in the calling sequence of actions where the loop is to be executed, an item containing the `Repeat` statement must be linked to the frame to be iterated. The logic to determine when the loop is complete must be placed at the appropriate point in the sequence of iterated actions. An `Exit` statement placed at this location terminates the loop and returns control to the calling frame. One benefit to this approach is that both programs and users use the same mechanism to access the referenced code.

In summary, KMS provides the ability to create arbitrarily complex databases of hypermedia frames. It provides operations to manipulate the frames. It also provides the capability to automate sequences of events via the KMS Action Language. Sample KMS frames from RSCRS are pictured in Appendix B.

Appendix B. *Sample Frames From RSCRS*

This appendix presents several sample frames from RSCRS. The first sequence of frames, Figures 9 through 11, traverses one branch of the classification tree. Note that each term on each of these frames contains a o `@Info` item which the user may select to obtain further information about the chosen term. Also, note the additional o @ items in Figure 11 which cross-reference to related information frames.

Reusable Software Component Representation System (RSCRS)

The text for the explanatory annotation items (indicated by @Info) is taken directly from the 1987 book, *Software Components With Ada* written by Grady Booch and published by The Benjamin/Cummings Publishing Company, Inc., at Menlo Park, California. Page references to the book are given in parentheses following the explanatory item.

Reusable Software Components ◦ @Info

- ◦ Structures ◦ @Info
- ◦ Tools ◦ @Info

Figure 9. Classification Tree — Top Level Frame

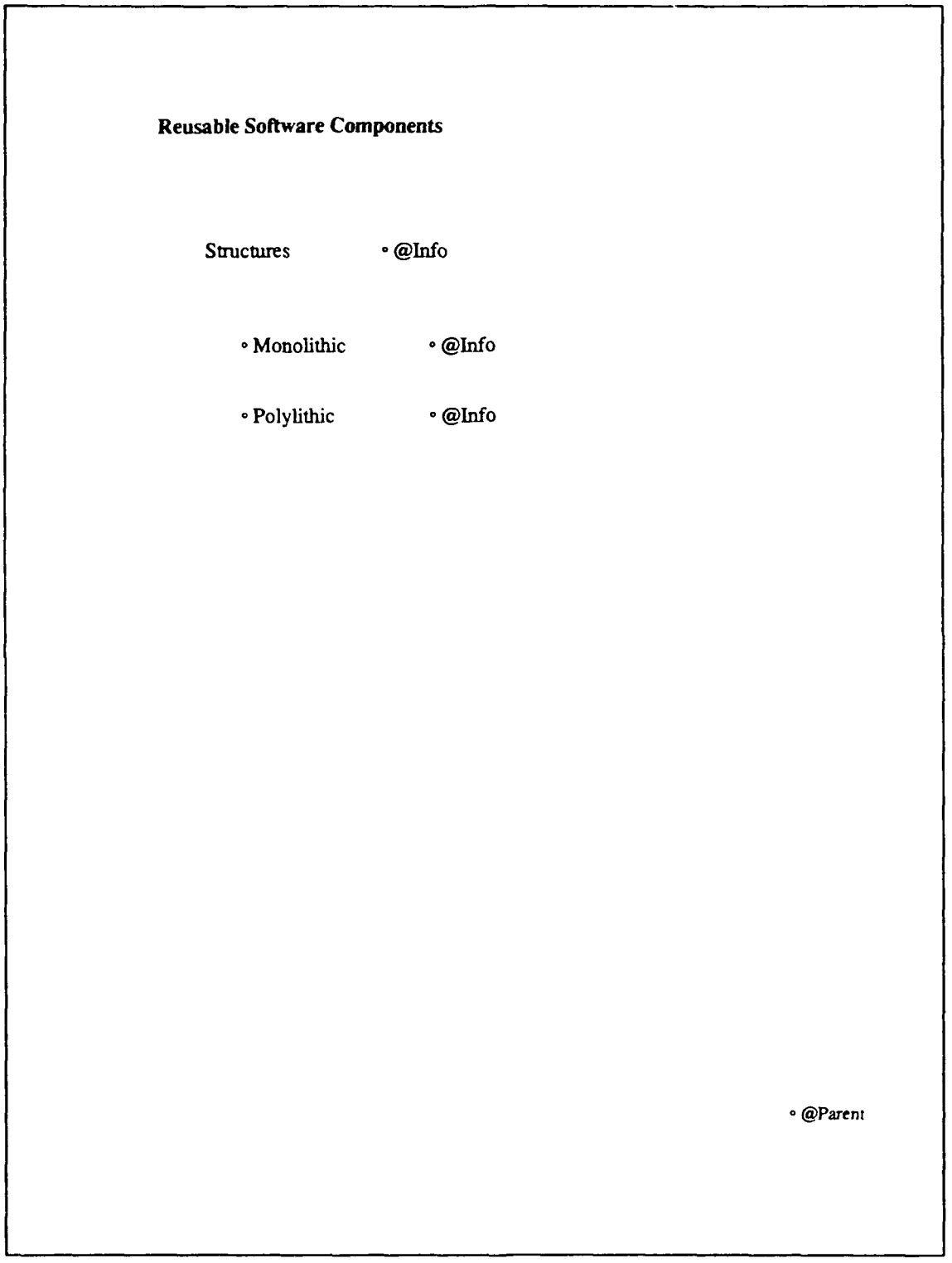


Figure 10. Classification Tree — Second Level Frame

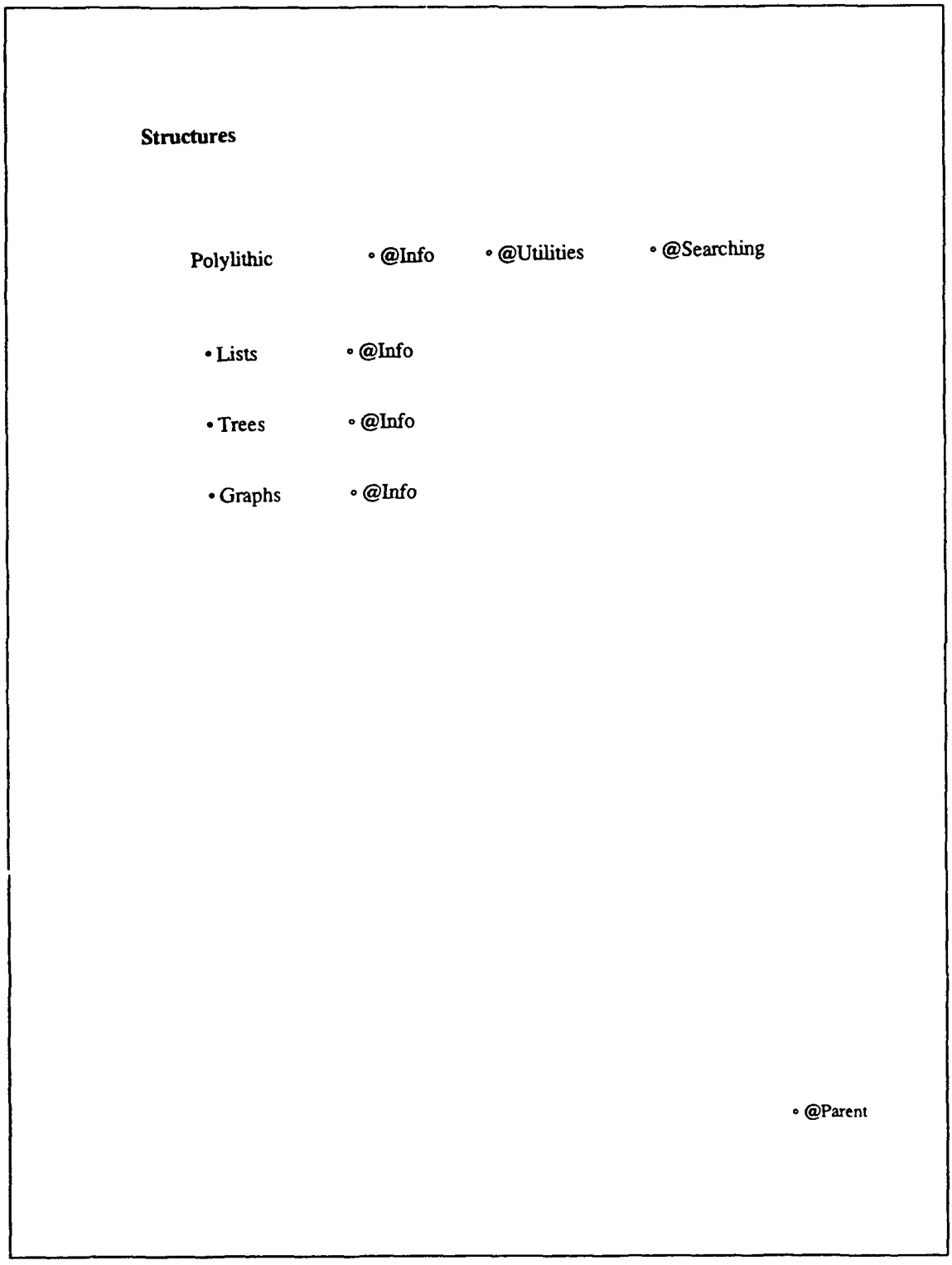


Figure 11. Classification Tree — Leaf Level Frame

The next frame, Figure 12, shows a sample explanatory text frame obtained by the user's selection of a o `@Info` item. This is followed by an abstract data type frame, Figure 13, and a time and space complexity table, Figure 14. These frames are obtained by selecting the appropriate items from the explanatory text frame.

Lists

◦ Abstract Data Type

◦ Time and Space Complexity Analysis

A *list* is a sequence of zero or more items in which items can be added and removed from any position such that a strict linear ordering is maintained. The type of the item is immaterial to the behavior of the list. We designate the ordering of items in a list by linking one item to the next.

◦ @Link Info

If a list contains zero items, we consider it to be *null*. If a list is not null, we call the first item the *head* of the list. The (possibly zero-length) sequence of items following the head is called the *tail* of the list. Since the tail itself is a sequence of items, the tail of a list is also a list. Because a list may contain subsequences that are themselves lists, we consider a list to be a polylithic component.

◦ @Polylithic Info

We may denote a list of length n as:

$$i_1, i_2, i_3, \ldots, i_n$$

The head of this list is the item i_1 and the tail is the list whose head is the item i_2. Furthermore, since there exists a linear ordering of each item i_i, we note that i_i precedes i_{i+1}.

(pp. 71-72)

◦ @Parent

Figure 12. Explanatory Text Frame

List Abstract Data Type

DEFINITION	A sequence of zero or more items			
VALUES	An ordered collection of items			
CONSTANTS	Null_List			
OPERATIONS	Constructors	Selectors	Iterator	Exceptions
	Share Copy Clear Construct Set_Head Swap_Tail	Is_Shared Is_Equal Length_Of Is_Null Head_Of Tail_Of Predecessor_Of		Overflow List_Is_Null Not_At_Head
FORMS (8)	Single/Double Unbounded/Bounded Unmanaged/Managed/Controlled			

(p. 78)

◦ @Parent

Figure 13. Abstract Data Type Frame

List Time and Space Complexity Analysis

DIMENSION		VALUE	ALTERNATE VALUE
$S(n)$		$O(n)$	$O(The\_Size)$ for bounded forms
$T(n)$	Share	$O(1)$	
	Copy	$O(n)$	
	Clear	$O(1)$	$O(n)$ for managed and controlled forms
	Construct	$O(1)$	
	Set_Head	$O(1)$	
	Swap_Tail	$O(1)$	
	Is_Shared	$O(1)$	
	Is_Equal	$O(Min(m,n))$	$O(1)$ for bounded forms
	Length_Of	$O(n)$	
	Is_Null	$O(1)$	
	Head_Of	$O(1)$	
	Tail_Of	$O(1)$	
	Predecessor_Of	$O(1)$	

(p. 102)

◦ @Parent

Figure 14. Time And Space Complexity Analysis Frame

Figure 15 shows a sample forms selection frame obtained by selecting one of the leaf items in the classification tree. Figure 16 shows the same frame after the forms selections have been made (indicated by **bold face** type).

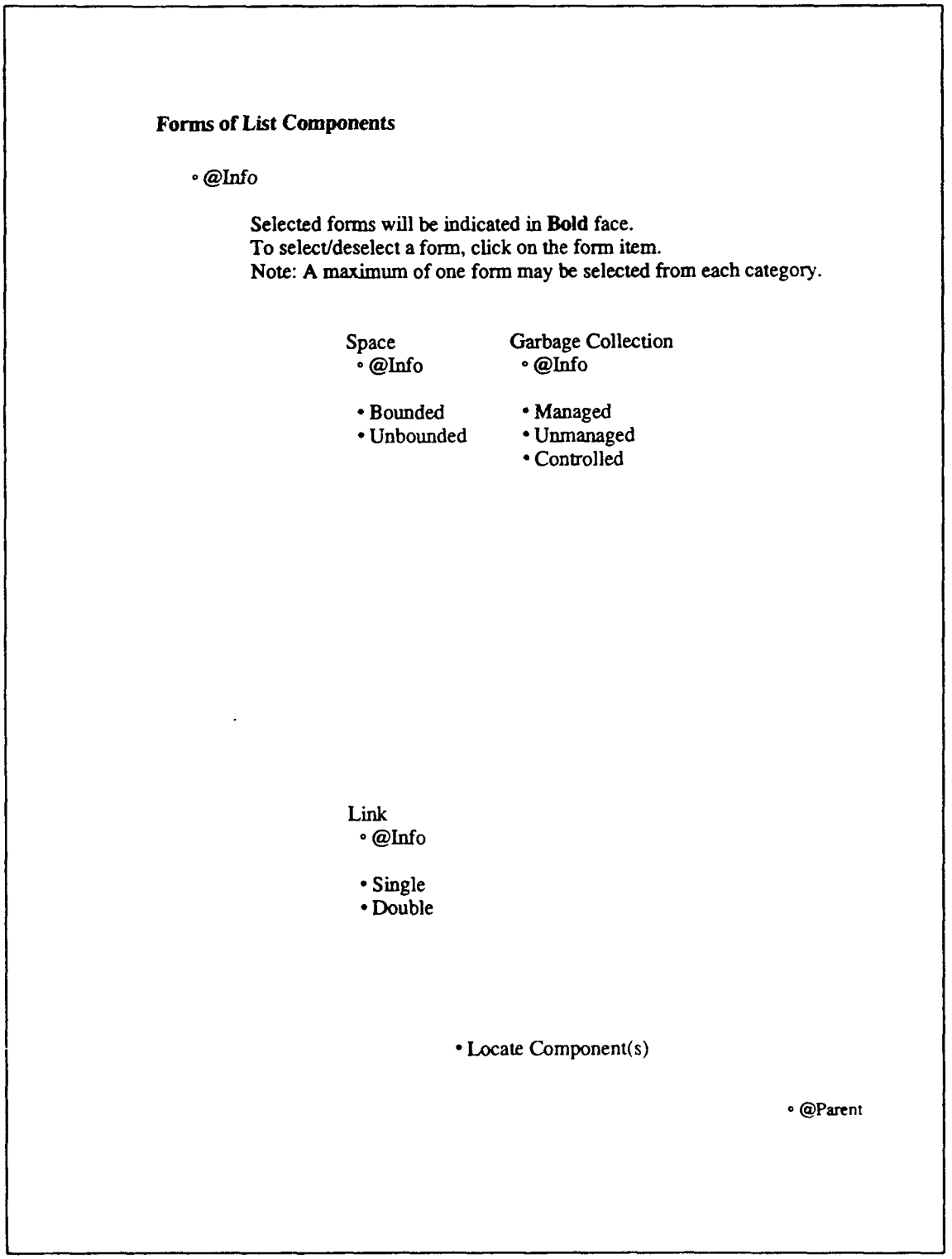


Figure 15. Forms Frame

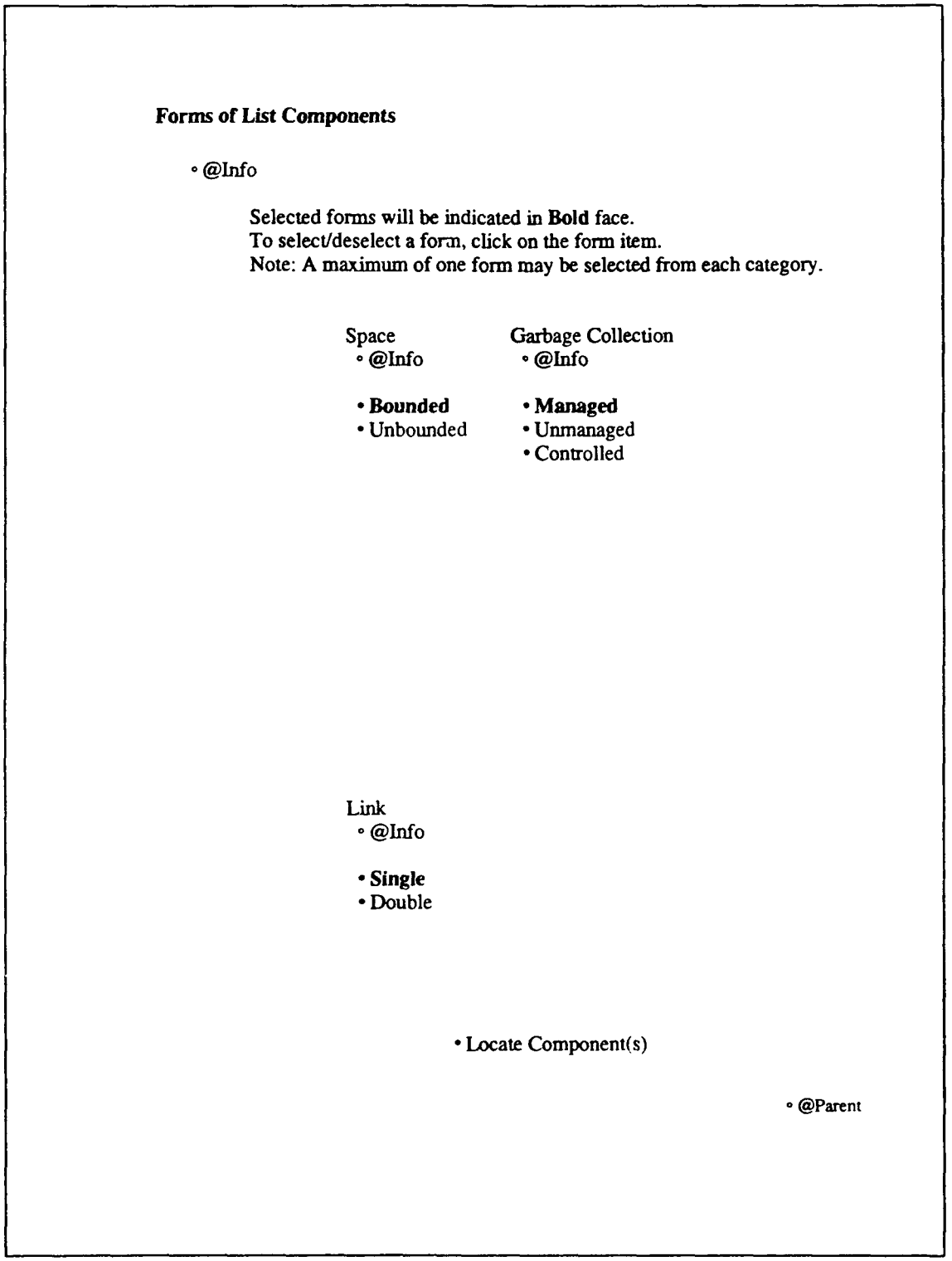


Figure 16. Forms Frame With Forms Selected

Figure 17 shows the selected component location information. This frame is displayed when the user selects the • `Locate Component(s)` item on the forms frame. Figure 18 shows the frame displayed when the user selects the • `Browse Spec File` item on the component location frame.

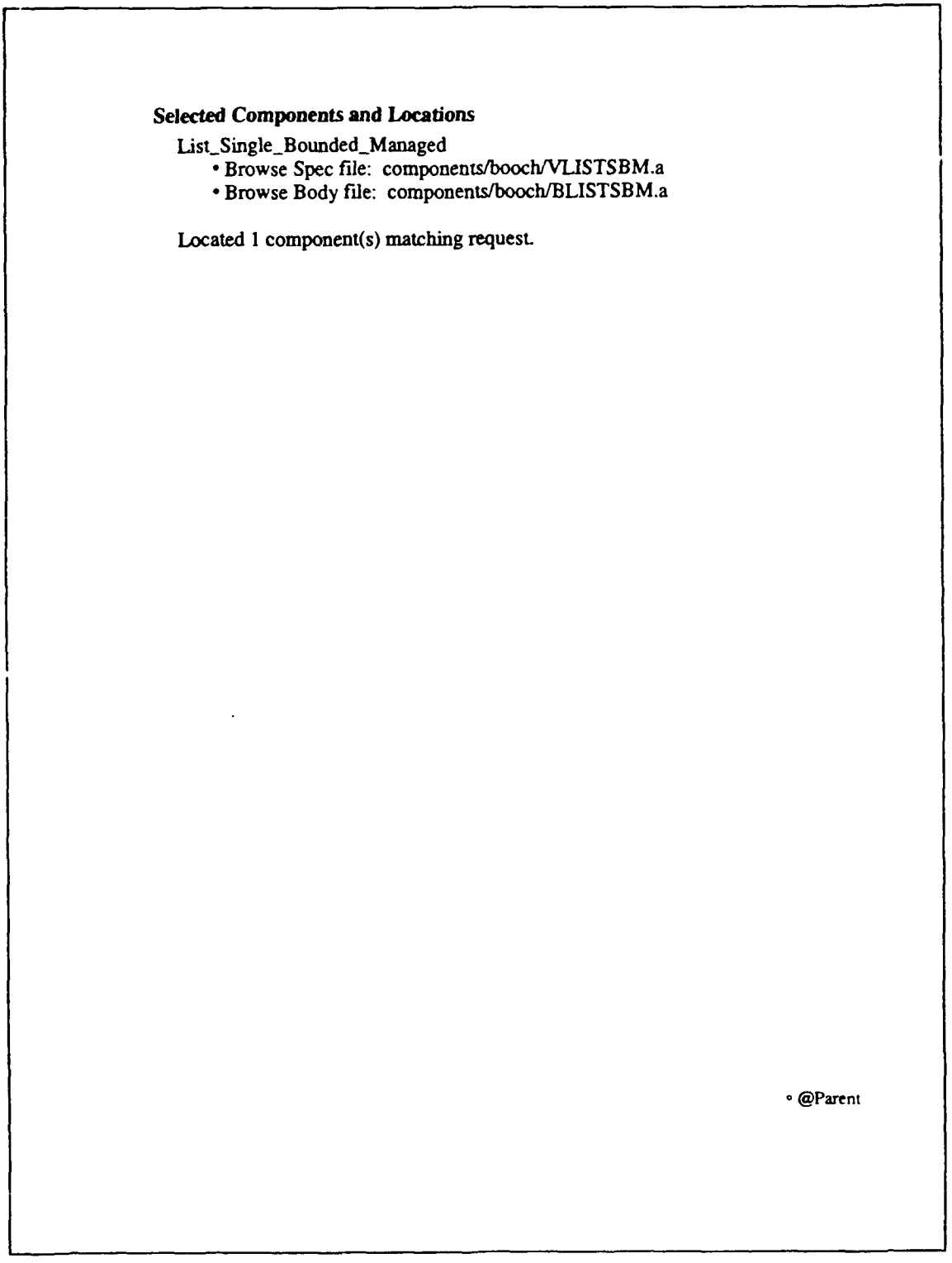


Figure 17. Component Location Frame

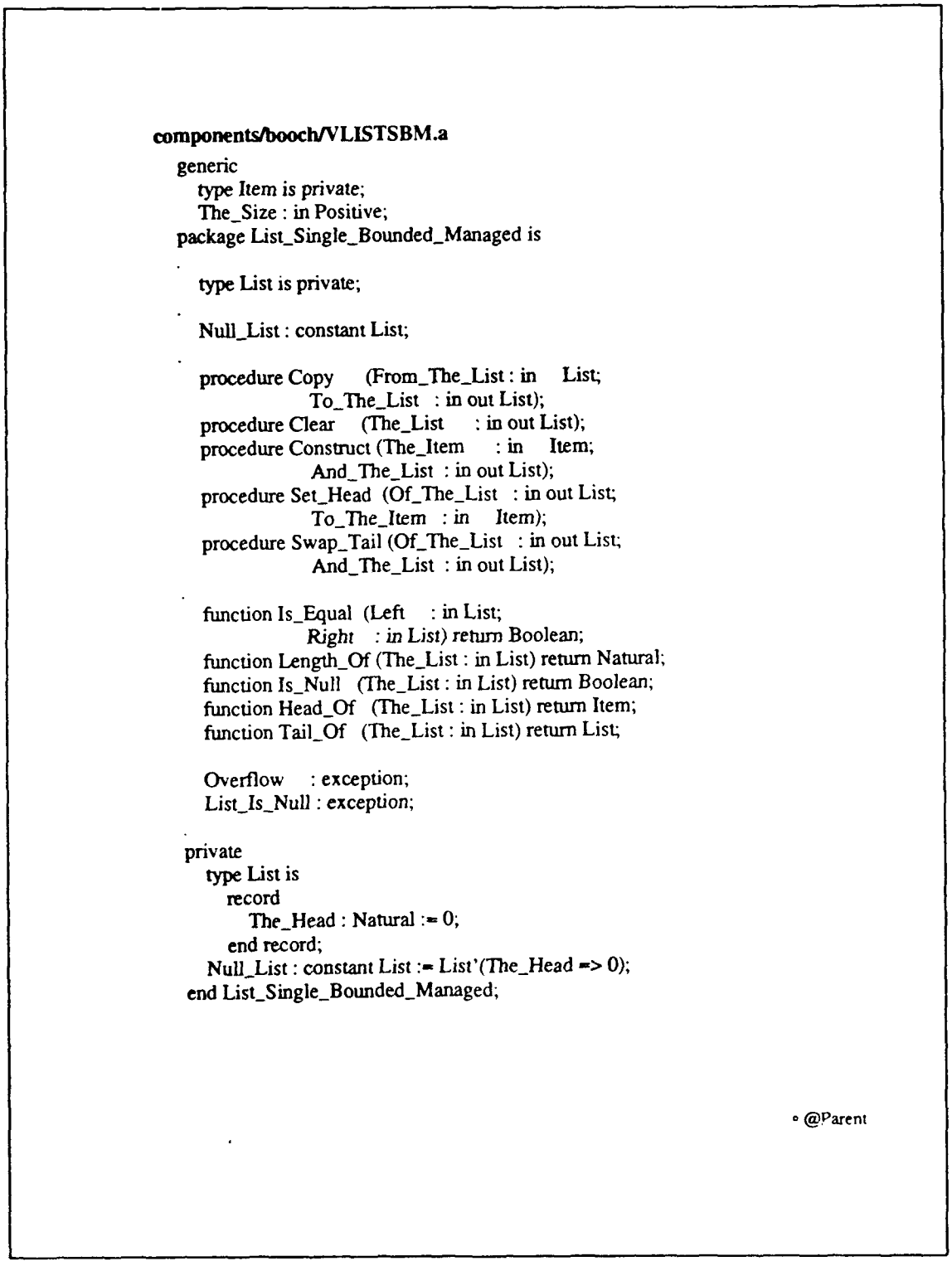


Figure 18. Browse Frame (contents from Booch, 1987:85-86)

Figure 19 shows the attributes of the Lists item on the Polylithic Structures selection frame. As explained in Appendix A, items preceded by a • symbol will execute an action when they are selected. Notice the Action attribute executes the CompForms9 frame, Figure 20.

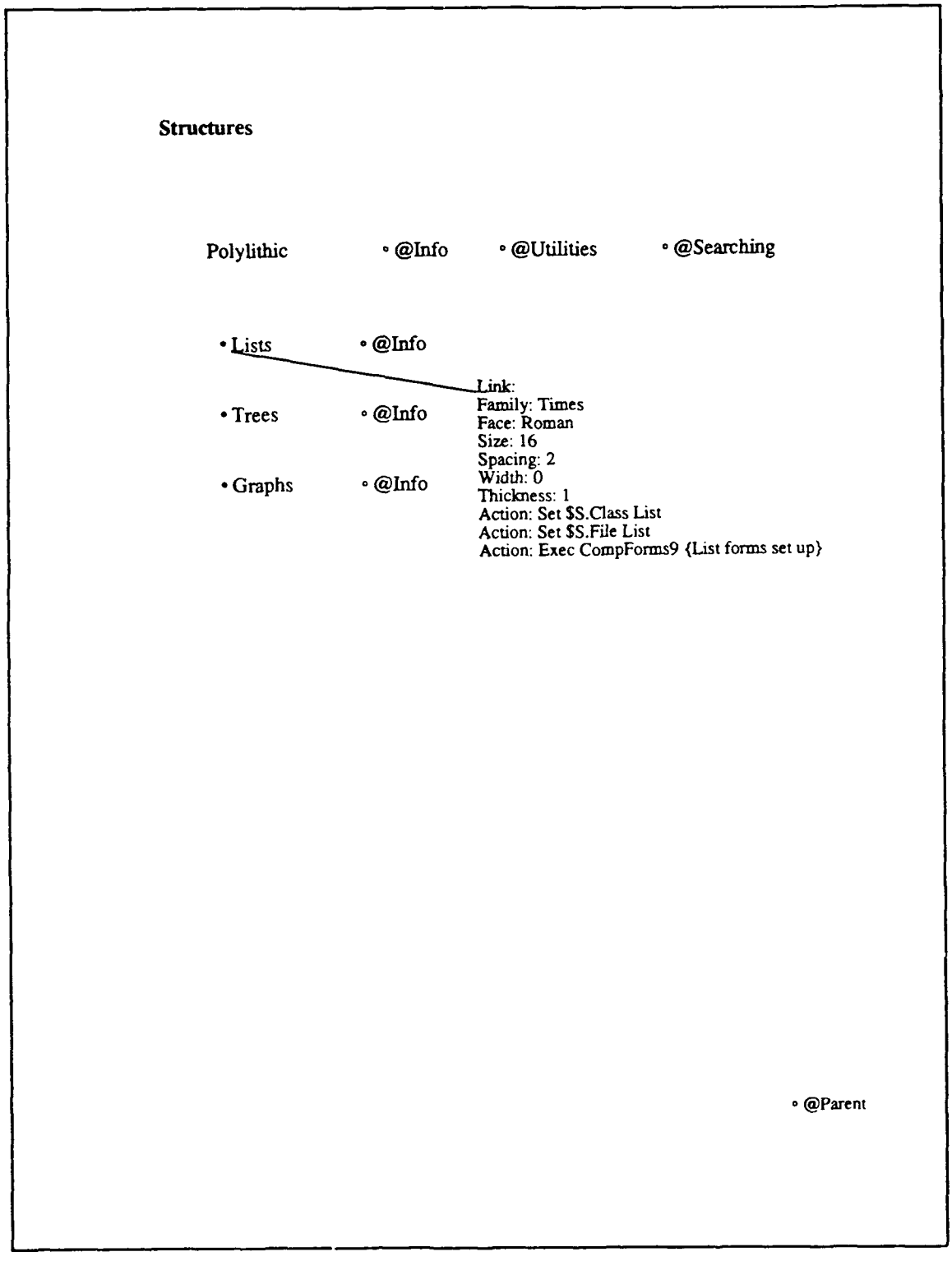


Figure 19. Item Attributes Example

The remaining figures, Figures 20 through 30 illustrate the use of the KMS Action Language within RSCRS. These figures contain the frames which are processed to generate the Forms frame applicable to List structures. Recall from Appendix A that transfers of control are accomplished via links in the same way that frames are traversed when viewing them. Notice that a link on a comment item (an item enclosed in curly braces, { and }) effectively makes that item serve as a procedure call. Also, the frame names which appear as comments are just that, comments. It is the Link attribute of the item that determines which frame will be executed.

As a convenience to anyone trying to read the KMS Action Language frames in RSCRS, all linked items are commented with the name of the frame being linked to. This saves having to extract the attribute information simply to read the program.

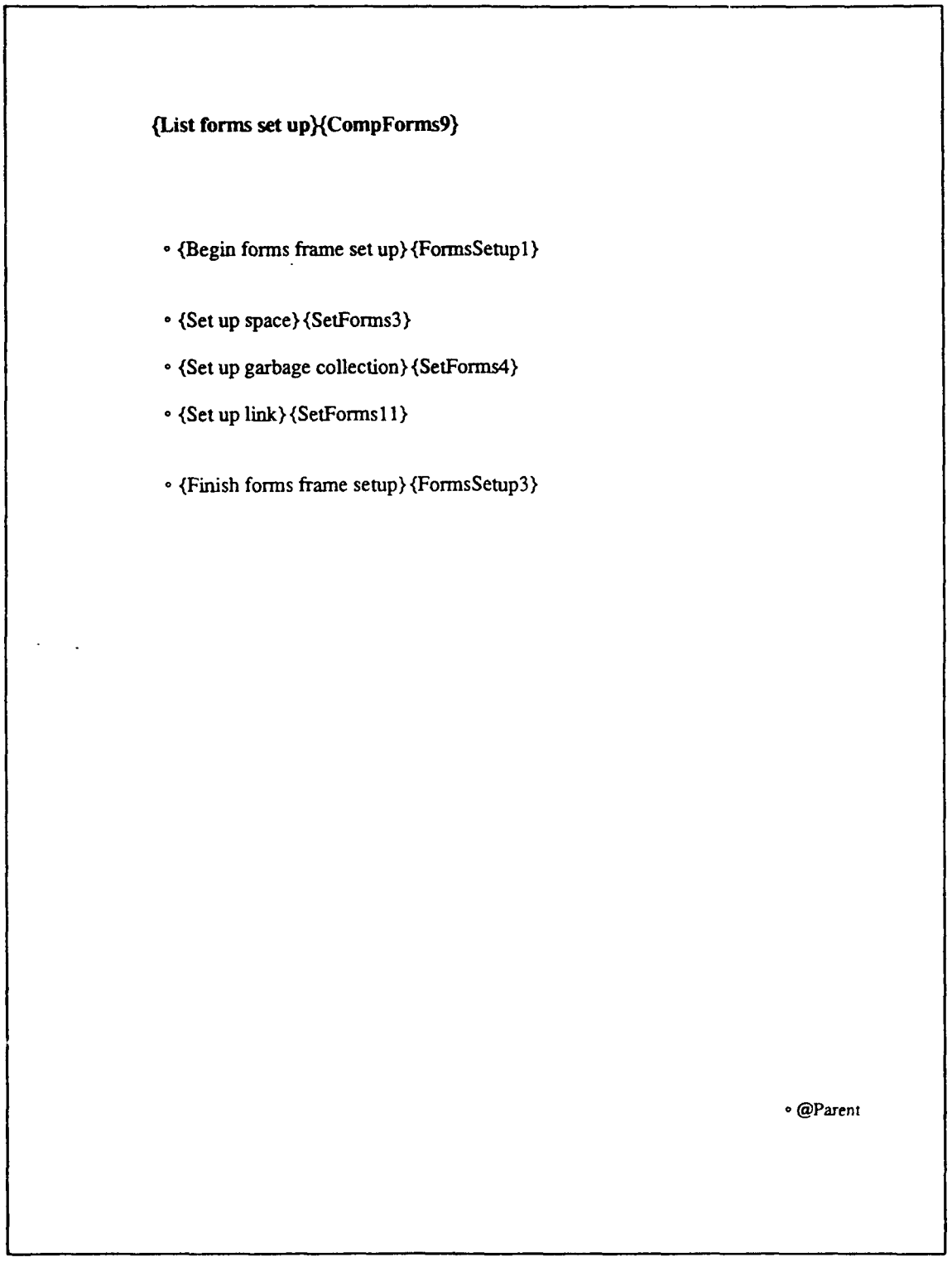


Figure 20. KMS Program Example

```
{Begin forms frame set up} {FormsSetup1}

    MessageLn " "
    MessageLn "Please wait -- creating forms frame"

    CurrentOpenFrame $FP.TaxonomyFrame
    GetFrameName $FP.TaxonomyFrame $S.FrameName

    OpenFrame Forms1 $FP.FormsFrame False $B.OpenStatus
    Set $B.CreateStatus True
    IfNot $B.OpenStatus CreateFrame Forms $FP.FormsFrame $B.CreateStatus
    IfNot $B.CreateStatus MessageLn "Unable to create Forms frame"
    IfNot $B.CreateStatus ExitAll

    GetFrameName $FP.FormsFrame $S.FormsFrame
    IfNotEqCase $S.FormsFrame Forms1 MessageLn "Frame name is not Forms1 -- Fix Forms frameset"
    IfNotEqCase $S.FormsFrame Forms1 CloseFrame $FP.FormsFrame
    IfNotEqCase $S.FormsFrame Forms1 ExitAll

    ClearFrame $FP.FormsFrame

  ◦ {Create static text on forms frame} {FormsSetup2}

  ◦ {Set up coordinate variables} {CoordSet1}
```

Figure 21. KMS Program Example (Continued)

```
{Create static text on forms frame}{FormsSetup2}

    GetTitle $FP.FormsFrame $IP.Title

    IfNotNull $IP.Title RemoveItemFromFrame $FP.FormsFrame $IP.Title

    ConcatStr "Forms of " $S.Class " Components" $S.Title

    CreateTitle $FP.FormsFrame $S.Title $IP.Title

    SetItemFace $IP.Title Bold

    UpdateItemRectangle $IP.Title

    CreateItem $FP.FormsFrame 36 54 $IP.Info "@Info"

    SetItemLink $IP.Info BTInfo98

    CreateItem $FP.FormsFrame 72 90 $IP.Inst1 "Selected forms will be indicated in Bold face."

    CreateItem $FP.FormsFrame 72 108 $IP.Inst2 "To select/deselect a form, click on the form item."

    CreateItem $FP.FormsFrame 72 126 $IP.Inst3 "Note: A maximum of one form may be selected from each category."

                                                                    ◦ @Parent
```

Figure 22. KMS Program Example (Continued)

{Set up coordinate variables}{CoordSet1}

{General forms}

Set $I.SequentialX 36
Set $I.SequentialY 234

Set $I.GuardedX 36
Set $I.GuardedY 252

Set $I.ConcurrentX 36
Set $I.ConcurrentY 270

Set $I.MultipleX 36
Set $I.MultipleY 288

Set $I.BoundedX 162
Set $I.BoundedY 234

Set $I.UnboundedX 162
Set $I.UnboundedY 252

Set $I.ManagedX 288
Set $I.ManagedY 234

Set $I.UnmanagedX 288
Set $I.UnmanagedY 252

Set $I.ControlledX 288
Set $I.ControlledY 270

Set $I.IteratorX 432
Set $I.IteratorY 234

Set $I.NoniteratorX 432
Set $I.NoniteratorY 252

- {Set up coordinate vars for special monolithic forms}{CoordSet2}
- {Set up coordinate vars for special polylithic forms}{CoordSet3}

Figure 23. KMS Program Example (Continued)

```
{Set up coordinate vars for special monolithic forms}{CoordSet2}
   Set $I.BalkingX 36
   Set $I.BalkingY 414

   Set $I.NonbalkingX 36
   Set $I.NonbalkingY 432


   Set $I.PriorityX 162
   Set $I.PriorityY 414

   Set $I.NonpriorityX 162
   Set $I.NonpriorityY 432


   Set $I.SimpleX 288
   Set $I.SimpleY 414

   Set $I.DiscreteX 288
   Set $I.DiscreteY 432


   Set $I.CachedX 432
   Set $I.CachedY 414

   Set $I.NoncachedX 432
   Set $I.NoncachedY 432

                                                    ◦ @Parent
```

Figure 24. KMS Program Example (Continued)

```
{Set up coordinate vars for special polylithic forms}{CoordSet3}
   Set $I.SingleX 162
   Set $I.SingleY 594

   Set $I.DoubleX 162
   Set $I.DoubleY 612


   Set $I.BinaryX 288
   Set $I.BinaryY 594

   Set $I.ArbitraryX 288
   Set $I.ArbitraryY 612


   Set $I.DirectedX 432
   Set $I.DirectedY 594

   Set $I.UndirectedX 432
   Set $I.UndirectedY 612

                                                    ◦ @Parent
```

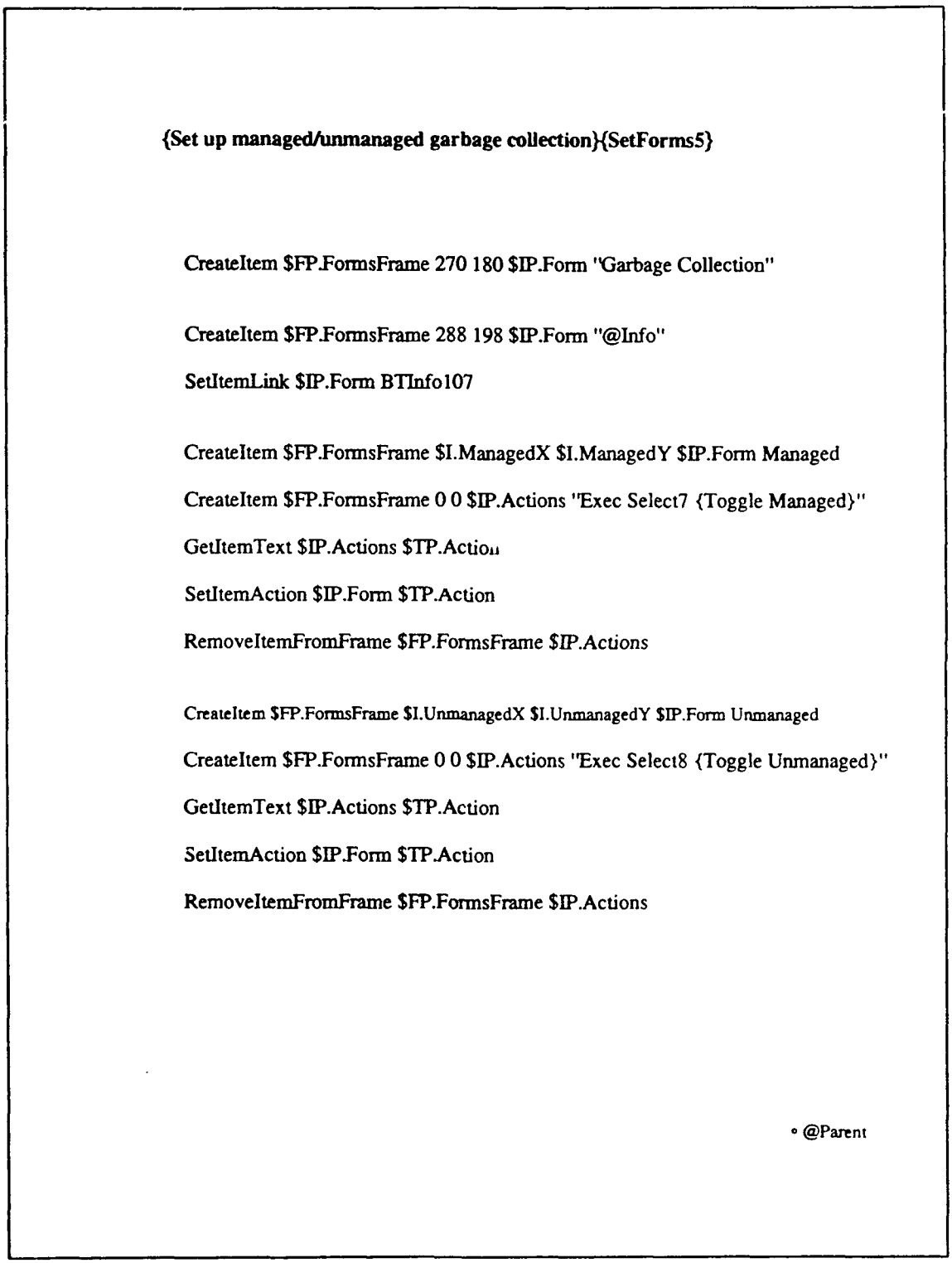
Figure 25. KMS Program Example (Continued)

```
{Set up space}{SetForms3}


    CreateItem $FP.FormsFrame 144 180 $IP.Form Space

    CreateItem $FP.FormsFrame 162 198 $IP.Form "@Info"
    SetItemLink $IP.Form BTInfo105

    CreateItem $FP.FormsFrame $I.BoundedX $I.BoundedY $IP.Form Bounded
    CreateItem $FP.FormsFrame 0 0 $IP.Actions "Exec Select5 {Toggle Bounded}"
    GetItemText $IP.Actions $TP.Action
    SetItemAction $IP.Form $TP.Action
    RemoveItemFromFrame $FP.FormsFrame $IP.Actions

    CreateItem $FP.FormsFrame $I.UnboundedX $I.UnboundedY $IP.Form Unbounded
    CreateItem $FP.FormsFrame 0 0 $IP.Actions "Exec Select6 {Toggle Unbounded}"
    GetItemText $IP.Actions $TP.Action
    SetItemAction $IP.Form $TP.Action
    RemoveItemFromFrame $FP.FormsFrame $IP.Actions
```

Figure 26. KMS Program Example (Continued)

```
{Set up garbage collection}{SetForms4}

  ◦ {Set up managed/unmanaged garbage collection}{SetForms5}

    CreateItem $FP.FormsFrame $I.ControlledX $I.ControlledY $IP.Form Controlled

    CreateItem $FP.FormsFrame 0 0 $IP.Actions "Exec Select9 {Toggle Controlled}"

    GetItemText $IP.Actions $TP.Action

    SetItemAction $IP.Form $TP.Action

    RemoveItemFromFrame $FP.FormsFrame $IP.Actions
```

Figure 27. KMS Program Example (Continued)



```
{Set up managed/unmanaged garbage collection}{SetForms5}

CreateItem $FP.FormsFrame 270 180 $IP.Form "Garbage Collection"

CreateItem $FP.FormsFrame 288 198 $IP.Form "@Info"
SetItemLink $IP.Form BTInfo107

CreateItem $FP.FormsFrame $I.ManagedX $I.ManagedY $IP.Form Managed
CreateItem $FP.FormsFrame 0 0 $IP.Actions "Exec Select7 {Toggle Managed}"
GetItemText $IP.Actions $TP.Action
SetItemAction $IP.Form $TP.Action
RemoveItemFromFrame $FP.FormsFrame $IP.Actions

CreateItem $FP.FormsFrame $I.UnmanagedX $I.UnmanagedY $IP.Form Unmanaged
CreateItem $FP.FormsFrame 0 0 $IP.Actions "Exec Select8 {Toggle Unmanaged}"
GetItemText $IP.Actions $TP.Action
SetItemAction $IP.Form $TP.Action
RemoveItemFromFrame $FP.FormsFrame $IP.Actions

                                                    ° @Parent
```

Figure 28. KMS Program Example (Continued)

```
{Set up link}{SetForms11}

    CreateItem $FP.FormsFrame 144 540 $IP.Form Link

    CreateItem $FP.FormsFrame 162 558 $IP.Form "@Info"
    SetItemLink $IP.Form BTInfo120

    CreateItem $FP.FormsFrame $I.SingleX $I.SingleY $IP.Form Single
    CreateItem $FP.FormsFrame 0 0 $IP.Actions "Exec Select20 {Toggle Single}"
    GetItemText $IP.Actions $TP.Action
    SetItemAction $IP.Form $TP.Action
    RemoveItemFromFrame $FP.FormsFrame $IP.Actions

    CreateItem $FP.FormsFrame $I.DoubleX $I.DoubleY $IP.Form Double
    CreateItem $FP.FormsFrame 0 0 $IP.Actions "Exec Select21 {Toggle Double}"
    GetItemText $IP.Actions $TP.Action
    SetItemAction $IP.Form $TP.Action
    RemoveItemFromFrame $FP.FormsFrame $IP.Actions

                                                    ◦ @Parent
```

Figure 29. KMS Program Example (Continued)

```
{Finish forms frame set up}{FormsSetup3}

    CreateItem $FP.FormsFrame 234 720 $IP.Form "Locate Component(s)"

    ConcatStr "Set $S.ListFile " $S.File $S.Set
    CreateItem $FP.FormsFrame 0 0 $IP.Actions $S.Set
    ConcatStr "Set $S.Class " $S.Class $S.Set
    AddItemText $IP.Actions $S.Set
    AddItemText $IP.Actions "Exec Locate1"

    GetItemText $IP.Actions $TP.Action
    SetItemAction $IP.Form $TP.Action

    RemoveItemFromFrame $FP.FormsFrame $IP.Actions

    CreateItem $FP.FormsFrame 488 767 $IP.Form "@Parent"
    SetItemLink $IP.Form $S.FrameName
    SetItemSize $IP.Form 14
    UpdateItemRectangle $IP.Form

    CloseFrame $FP.FormsFrame

    ClearMessage

    Goto Forms1
```

Figure 30. KMS Program Example (Continued)

Bibliography

Anderson, Chris, Project Manager. "Software Reuse: A CAMP Project Update" Paper presented at the AIAA Missile Science Conference, Monterey, California, 29 November - 1 December 1988.

Arnold, Susan P. and Stephen L. Stepoway. "The Reuse System: Cataloguing and Retrieval of Reusable Software," *COMPCON Spring 87 digest of papers.* 376–379. Washington, D.C.: Computer Society Press of the IEEE, 1987.

Biggerstaff, Ted J. *Hypermedia as a Tool to Aid Large Scale Reuse.* MCC Technical Report Number STP-202-87. Austin, Texas: Microelectronics and Computer Technology Corporation, July 1987.

Biggerstaff, Ted J. and Alan J. Perlis. "Foreword," *IEEE Transactions on Software Engineering, SE-10*: 474–477 (September 1984).

Biggerstaff, Ted and Charles Richter. "Reusability Framework, Assessment, and Directions," *IEEE Software, 4*: 41–49 (March 1987).

Booch, Grady. *Software Components With Ada.* Menlo Park, California: Benjamin/Cummings Publishing Company, Inc., 1987.

Boyle, James M. and Monagur N. Muralidharan. "Program Reusability through Program Transformation," *IEEE Transactions on Software Engineering, SE-10*: 574-588 (September 1984).

Buchanan, B. *Theory of Library Classification.* New York: K. G. Saur Publishing, Inc., 1979.

Burton, Bruce A. and others. "The Reusable Software Library," *IEEE Software*: 25–33 (July 1987).

Cardow, Capt James E. "Issues on Software Reuse," *Proceedings of The IEEE 1989 National Aerospace and Electronics Conference NAECON 1989, 2.* 564–570. New York: IEEE Press, 1989.

Chan, L. M. *Cataloging and Classification.* New York: McGraw-Hill Book Company, 1981.

Cheatham, Thomas E., Jr. "Reusability Through Program Transformations," *IEEE Transactions on Software Engineering, SE-10*: 589-594 (September 1984).

Devanbu, Premkumar and others. "A Knowledge-Based Software Information System," *Eleventh International Joint Conference on Artificial Intelligence Proceedings, 1.* 110–115. San Mateo, California: Morgan Kaufmann Publishers, Inc., 1989.

Frakes, W. B. and P. B. Gandel. "Representation Methods for Software Reuse," *TRI-Ada '89 Conference Proceedings.* 302–314. New York: Association for Computing Machinery, 1989.

Frakes, W. B. and B. A. Nejmeh. "Software Reuse Through Information Retrieval," *COMPCON Spring 87 digest of papers.* 380–384. Washington, D.C.: Computer Society Press of the IEEE, 1987.

Freeman, Peter. "A Conceptual Analysis of the Draco Approach to Constructing Software Systems," *IEEE Transactions on Software Engineering, SE-13*: 830–850 (July 1987).

Garey, Michael R. and David S. Johnson. *Computers and Intractability.* San Francisco: W. H. Freeman and Company, 1979.

Getting Started with KMS. Version 11. Knowledge Systems, Murrysville, Pennsylvania, August 1988.

Ghisio, O. and others. "An Extended Approach to Reusability," *COMPCON Spring 87 digest of papers.* 385–389. Washington, D.C.: Computer Society Press of the IEEE, 1987.

Gibbs, Simon and others. "Muse: A Multimedia Filing System," *IEEE Software, 4*: 4–15 (March 1987).

Horowitz, Ellis and John B. Munson. "An Expansive View of Reusable Software," *IEEE Transactions on Software Engineering, SE-10*: 477–487 (September 1984).

Introduction to KMS. Version 17. Knowledge Systems, Murrysville, Pennsylvania, August 1988.

Jones, T. Capers. "Reusability in Programming: A Survey of the State of the Art," *IEEE Transactions on Software Engineering, SE-10*: 488–494 (September 1984).

KMS Action Language Manual. Version 8. Knowledge Systems, Murrysville, Pennsylvania, November 1988.

KMS Reference Manual. Version 7. Knowledge Systems, Murrysville, Pennsylvania, October 1988.

Knapper, Robert J. "An Introduction for the TRI-Ada Session on Reusability," *TRI-Ada '88 Conference Proceedings.* 232–236. New York: Association for Computing Machinery, 1988.

Lanergan, Robert G. and Charles A. Grasso. "Software Engineering with Reusable Designs and Code," *IEEE Transactions on Software Engineering, SE-10*: 498–501 (September 1984).

Latour, Larry and Elizabeth Johnson. "Seer: A Graphical Retrieval System for Reusable Ada Software Modules," *The Third International IEEE Conference on Ada Applications and Environments proceedings.* 105–113. Washington, D.C.: Computer Society Press, 1988.

Litvintchouk, Steven D. and Allen S. Matsumoto. "Design of Ada Systems Yielding Reusable Components: An Approach using Structured Algebraic Specification," *IEEE Transactions on Software Engineering, SE-10*: 544–551 (September 1984).

Margono, Johan and Edward V. Berard. "A Modified Booch's Taxonomy for Ada Generic Data-Structure Components and Their Implementation," *Ada Components: Libraries and Tools (Proceedings of the Ada-Europe International Conference Stockholm 26–28 May 87).* 61–74. New York: Cambridge University Press, 1987.

Meyer, Bertrand. "Reusability: The Case for Object-Oriented Design," *IEEE Software, 4*: 50–64 (March 1987).

Neighbors, James M. "The Draco Approach to Constructing Software from Reusable Components," *IEEE Transactions on Software Engineering, SE-10*: 564–574 (September 1984).

Perry, J. M. *Perspective on Software Reuse.* Technical Report CMU/SEI-88-TR-22. Pittsburgh: Software Engineering Institute, September 1988 (AD-A204399).

Prieto-Diaz, Rubén. *A Software Classification Scheme.* PhD dissertation. University of California, Irvine, California, 1985.

Prieto-Diaz, Rubén and Peter Freeman. "Classifying Software for Reusability," *IEEE Software, 4*: 6–16 (January 1987).

Rosales, S. Roody and Prem K. Mehrotra. "MES: An Expert System for Reusing Models of Transmission Equipment," *Proceedings, The Fourth Conference on Artificial Intelligence Applications.* 109–113. Washington, D.C.: Computer Society Press of the IEEE, 1988.

Ruble, Daniel Lee. *A Classification Methodology and Retrieval Model to Support Software Reuse.* PhD Dissertation. Texas A&M University, 1987 (AD-A196541).

Sommerville, Ian. *Software Engineering* (Third Edition). New York: Addison-Wesley Publishing Company, 1989.

Standish, T. A. "An Essay on Software Reuse," *IEEE Transactions on Software Engineering, SE-10*: 496 (September 1984).

Tracz, Will. "Software Reuse: Motivators and Inhibitors," *COMPCON Spring 87 digest of papers.* 358–363. Washington, D.C.: Computer Society Press of the IEEE, 1987.

Vickery, B. C. *Faceted Classification: A Guide to Construction and Use of Special Schemes.* London: Aslib, 1960.

Wald, Elizabeth E. "Software Engineering with Reusable Parts," *COMPCON Spring 87 digest of papers.* 353–356. Washington, D.C.: Computer Society Press of the IEEE, 1987.

Wegner, Peter. "Varieties of Reusability," *Proceedings, ITT Workshop on Reusability in Programming.* 30–44. Stratford, Connecticut: ITT Programming, 1983.

Wegner, Peter. "Capital-intensive Technology and Reusability," *IEEE Software, 1.* 7–45 (July 1984).

Williams, William F. *Principles of Automated Information Retrieval.* Elmhurst, Illinois: O. A. Business Publications, Inc., 1965.

Wood, Murray and Ian Sommerville. "An Information Retrieval System for Software Components," *Software Engineering Journal, 3*: 198–207 (September 1988).

Vita

Captain Gary Gerard Worrall was born to Joseph and Theresa Worrall on 10 June, 1953 in Bethesda, Maryland. He graduated from First Colonial High School in Virginia Beach, Virginia in June 1971. He received an A.A.S. degree in Data Processing from the Community College of the Air Force in April 1981 and a B.S.C.S. in Computer Science from the University of South Florida in April 1984. He married Andrea Katharine in July 1982. They have two sons, Colin and Graham. Captain Worrall enlisted in the United States Air Force in September 1976 and received technical training as a computer programmer at Keesler AFB, Mississippi. He then served as a Special Activity Computer Programming Specialist at Headquarters Tactical Air Command, Langley AFB, Virginia from January 1977 to August 1981 when he was selected for the Airman's Education and Commissioning Program. He received his commission as an officer in the United States Air Force in August 1984. He then served as Space Shuttle Launch Processing System Station Manager at Vandenberg AFB, California until June 1987. His next assignment was as a Computer Research Scientist at the Air Force Acquisition Logistics Center, Wright-Patterson AFB, Ohio. He pursued his M.S. in Computer Science at the Air Force Institute of Technology from May 1989 to December 1990. He may be contacted at the Human Resources Laboratory, Wright-Patterson AFB, Ohio 45433.

Permanent address: 1651 Gulf Blvd. #1-23
Clearwater, FL 34630

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December 1990	Master's Thesis

4. TITLE AND SUBTITLE
A HYPERMEDIA IMPLEMENTATION FOR REUSABLE SOFTWARE COMPONENT REPRESENTATION

5. FUNDING NUMBERS

6. AUTHOR(S)
Gary G. Worrall, Capt, USAF

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Air Force Institute of Technology, WPAFB OH 45433-6583

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT/GCS/ENG/90D-16

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This study investigated software component representation methods. Hypermedia was chosen as the implementation method to represent a collection of reusable software components. The hypermedia implementation organizes knowledge about the component collection into a web of small information chunks called frames.

The set of software components was represented using a hybrid classification scheme composed of enumerated and faceted parts. The enumerated part enables the user to progress along a path in a taxonomic tree, narrowing the scope of eligible components. Each leaf node in this tree denotes a class of components, members of which are distinguished by their time and space characteristics. These characteristics, known as forms, are grouped into eleven facets, each comprised of two to four elements.

Links between frames establish a means of traversing the information net. Some of these links allow the user to progress directly through the levels of the classification structure. Other links lead from the classification structure frames to frames containing explanatory text for the terms used in the classification. Additional links cross-reference related topics.

A simple component locator utilizes information from the frame selections to identify, and provide locating information for the desired components. A component source code browsing capability is provided.

14. SUBJECT TERMS
Software Engineering, Computer Programs, Ada Programming Language, Software Component Representation, Hypermedia

15. NUMBER OF PAGES
95

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL